Table of Contents

Chromium_doc_zh

Chromium中文文档 for

https://www.chromium.org/developers/design-documents

Design Documents

- Start Here: Background Reading: Describes the high-level architecture of Chromium
  - Multi-process Architecture

  Note: Most of the rest of the design documents assume familiarity with the concepts explained in this document.

  - How Chromium Displays Web Pages: Bottom-to-top overview of how WebKit is embedded in Chromium

See Also: Design docs in source code

```
https://chromium.googlesource.com/chromium/src/+/master/docs/
```

General Architecture

- General Architecture
  - Conventions and patterns for multi-platform development
  - Extension Security Architecture: How the extension system helps reduce the severity of extension vulnerabilities
  - HW Video Acceleration in Chrom{e,ium}{,OS}
  - Inter-process Communication: How the browser, renderer, and plugin processes communicate
  - Multi-process Resource Loading: How pages and images are loaded from the network into the renderer
  - Plugin Architecture
  - Process Models: Our strategies for creating new renderer processes
  - Profile Architecture
  - SafeBrowsing
  - Sandbox
  - Security Architecture: How Chromium's sandboxed rendering engine helps protect against malware
  - Startup
  - Threading: How to use threads in Chromium Also see the documentation for V8, which is the JavaScript engine used within Chromium.
- UI Framework
  - UI Development Practices: Best practices for UI development inside and outside of Chrome's content areas.
  - Views framework: Our UI layout layer used on Windows/Chrome OS.
  - views Windowing system: How to build dialog boxes and other windowed UI using views.
  - Aura: Chrome's next generation hardware accelerated UI framework, and the new ChromeOS window manager built using it.
  - NativeControls: using platform-native widgets in views.
  - Focus and Activation with Views and Aura.
- Graphics
  - Overview
  - GPU Accelerated Compositing in Chrome
  - GPU Feature Status Dashboard
  - Rendering Architecture Diagrams

  - Graphics and Skia
  - RenderText and Chrome UI text drawing
  - GPU Command Buffer
  - GPU Program Caching
  - Compositing in Blink/WebCore
  - Compositor Thread Architecture
  - Rendering Benchmarks
  - Impl-side Painting
  - Video Playback and Compositor
  - ANGLE architecture presentation
- Network stack
  - Overview
  - Network Stack Objectives
  - Crypto
  - Disk Cache
  - HTTP Cache
  - Out of Process Proxy Resolving Draft [unimplemented]
  - Proxy Settings and Fallback
  - Debugging network proxy problems
  - HTTP Authentication
  - View network internals tool
  - Make the web faster with SPDY pages
  - the web even faster with QUIC pages
  - Cookie storage and retrieval
- Security
  - Security Overview
  - Protecting Cached User Data
  - System Hardening
  - Chaps Technical Design
  - TPM Usage
  - Per-page Suborigins
  - Encrypted Partition Recovery
- Input
  - Seechromium inputfor design docs and other resources.
- Rendering
  - Multi-column layout
  - Style Invalidation in Blink
  - Blink Coordinate Spaces
- Building
  - IDL build
  - IDL compiler
  - See also the documentation for GYP, the build script generation tool.
- Testing
  - Layout test results dashboard
  - Generic theme for Test Shell
  - Moving LayoutTests fully upstream
- Feature-Specific
  - about:conflicts
  - Accessibility: An outline of current (and coming) accessibility support.
  - Auto-Throttled Screen Capture and Mirroring
  - Browser Window
  - Chromium Print Proxy: Enables a cloud print service for legacy printers and future cloud-aware printers.
  - Constrained Popup Windows

- Desktop Notifications
- DirectWrite Font Cache for Chrome on Windows
- DNS Prefetching: Reducing perceived latency by resolving domain names before a user tries to follow a link
- Embedding Flash Fullscreen in the Browser Window
- Extensions: Design documents and proposed APIs. : Design documents and proposed APIs.
- Find Bar
- Form Autofill: A feature to automatically fill out an html form with appropriate data.
- Geolocation: Adding support for W3C Geolocation API using native WebKit bindings.
- IDN in Google Chrome
- IndexedDB (early draft)
- Info Bars
- Installer: Registry entries and shortcuts
- Instant
- Isolated Sites
- Linux Resources and Localized Strings: Loading data resources and localized strings on Linux.
- Media Router & Web Presentation API
- Memory Usage Backgrounder: Some information on how we measure memory in Chromium.
- Mouse Lock
- Omnibox Autocomplete: While typing into the omnibox, Chromium searches for and suggests possible completions.
    - HistoryQuickProvider: Suggests completions from the user's historical site visits.
- Omnibox/IME Coordination
- Ozone Porting Abstraction
- Password Generation
- Pepper plugin implementation
- Plugin Power Saver
- Preferences
- Prerender
- Print Preview
- Printing
- Rect-based event targeting in views: Making it easier to target views elements with touch.
- Replace the modal cookie prompt
- SafeSearch
- Sane Time: Determining an accurate time in Chrome
- Secure Web Proxy
- Service Processes
- Site Isolation: In-progress effort to improve Chromium's process model for security between web sites.
- Software Updates: Courgette
- Sync
- Tab Helpers
- Tab to search: How to have the Omnibox automatically provide tab to search for your site.
- Tabtastic2 Requirements
- Temporary downloads
- Time Sources: Determining the time on a Chrome OS device
- TimeTicks: How our monotonic timer, TimeTicks, works on different OSes
- UI Mirroring Infrastructure: Describes the UI framework in ChromeViews that allows mirroring the browser UI in RTL locales such as Hebrew and Arabic.
- UI Localization: Describes how localized strings get added to Chromium.
- User scripts: Information on Chromium's support for user scripts.
- Video
- WebSocket: Enables Web applications to maintain bidirectional communications with server-side processes.
- Web MIDI
- WebNavigation API internals

- OS-Specific
  - Android
    - Java Resources on Android
    - JNI Bindings
    - WebView code organization
  - Chrome OS
    - See the Chrome OS design documents section.section.
  - Mac OS X
    - AppleScript Support
    - BrowserWindowController Object Ownership
    - Confirm to Quit
    - Mac App Mode (Draft)
    - Mac Fullscreen Mode (Draft)
    - Mac NPAPI Plugin Hosting
    - Mac specific notes on UI Localization
    - Menus, Hotkeys, & Command Dispatch
    - Notes from meeting on IOSurface usage and semantics
    - OS X Interprocess Communication (Obsolete)
    - Password Manager/Keychain Integration
    - Sandboxing Design
    - Tab Strip Design (Includes tab layout and tab dragging)
    - Wrench Menu Buttons
- Other
  - 64-bit Support
  - Browser Components / Layered Components
  - Closure Compiling Chrome Code
  - content module / content API
  - Design docs that still need to be written (wiki)
  - In progress refactoring of key browser-process architecture for porting
  - Network Experiments
  - Transitioning InlineBoxes from floats to LayoutUnits

- Start Here: Background Reading: Describes the high-level architecture of Chromium
    - Multi-process Architecture

  Note: Most of the rest of the design documents assume familiarity with the concepts explained in this document.

    - How Chromium Displays Web Pages: Bottom-to-top overview of how WebKit is embedded in Chromium

Multi-process Architecture

This document describes Chromium's high-level architecture.

Problem

It's nearly impossible to build a rendering engine that never crashes or hangs. It's also nearly impossible to build a rendering engine that is perfectly secure.

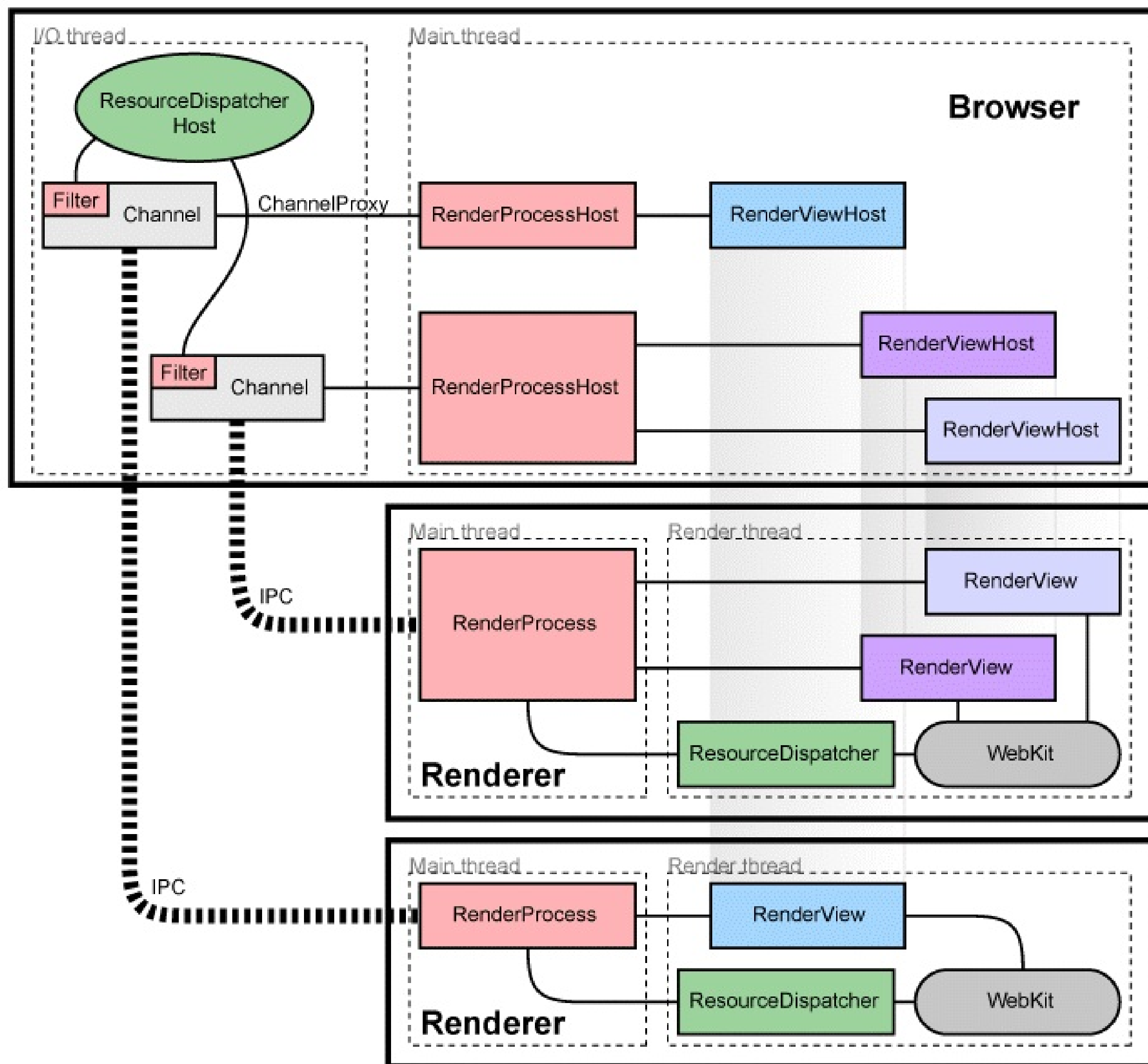
In some ways, the current state of web browsers is like that of the single-user, co-operatively multi-tasked operating systems of the past. As a misbehaving application in such an operating system could take down the entire system, so can a misbehaving web page in a modern web browser. All it takes is one browser or plug-in bug to bring down the entire browser and all of the currently running tabs.

Modern operating systems are more robust because they put applications into separate processes that are walled off from one another. A crash in one application generally does not impair other applications or the integrity of the operating system, and each user's access to other users' data is restricted.

Architectural overview

We use separate processes for browser tabs to protect the overall application from bugs and glitches in the rendering engine. We also restrict access from each rendering engine process to others and to the rest of the system. In some ways, this brings to web browsing the benefits that memory protection and access control brought to operating systems.

We refer to the main process that runs the UI and manages tab and plugin processes as the "browser process" or "browser." Likewise, the tab-specific processes are called "render processes" or "renderers." The renderers use the WebKit open-source layout engine for interpreting and laying out HTML.



Managing render processes

Each render process has a global RenderProcess object that manages communication with the parent browser process and maintains global state. The browser maintains a corresponding RenderProcessHost for each render process, which manages browser state and communication for the renderer. The browser and the renderers communicate using Chromium's IPC system.

Managing views

Each render process has one or more RenderView objects, managed by the RenderProcess, which correspond to tabs of content. The corresponding RenderProcessHost maintains a RenderViewHost corresponding to each view in the renderer. Each view is given a view ID that is used to differentiate multiple views in the same renderer. These IDs are unique inside one renderer but not within the browser, so identifying a view requires a RenderProcessHost and a view ID. Communication from the browser to a specific tab of content is done through these RenderViewHost objects, which know how to send messages through their RenderProcessHost to the RenderProcess and on to the RenderView.

Components and interfaces

In the render process:

- The *RenderProcess* handles IPC with the corresponding *RenderProcessHost* in the browser. There is exactly one RenderProcess object per render process. This is how all browser ↔ renderer communication happens.
- The *RenderView* object communicates with its corresponding *RenderViewHost* in the browser process (via the RenderProcess), and our WebKit embedding layer. This object represents the contents of one web page in a tab or popup window

In the browser process:

- The Browser object represents a top-level browser window.
- The RenderProcessHost object represents the browser side of a single browser ↔ renderer IPC connection. There is one RenderProcessHost in the browser process for each render process.
- The RenderViewHost object encapsulates communication with the remote RenderView, and RenderWidgetHost handles the input and painting for RenderWidget in the browser.

For more detailed information on how this embedding works, see the How Chromium displays web pages design document.

Sharing the render process

In general, each new window or tab opens in a new process. The browser will spawn a new process and instruct it to create a single *RenderView*.

Sometimes it is necessary or desirable to share the render process between tabs or windows. A web application opens a new window that it expects to communicate with synchronously, for example, using window.open in JavaScript. In this case, when we create a new window or tab, we need to reuse the process that the window was opened with. We also have strategies to assign new tabs to existing processes if the total number of processes is too large, or if the user already has a process open navigated to that domain. These strategies are described in Process Models.

Detecting crashed or misbehaving renderers

Each IPC connection to a browser process watches the process handles. If these handles are signaled, the render process has crashed and the tabs are notified of the crash. For now, we show a "sad tab" screen that notifies the user that the renderer has crashed. The page can be reloaded by pressing the reload button or by starting a new navigation. When this happens, we notice that there is no process and create a new one.

Sandboxing the renderer

Given Webkit is running in a separate process, we have the opportunity to restrict its access to system resources. For example, we can ensure that the renderer's only access to the network is via its parent browser process. Likewise, we can restrict its access to the filesystem using the host operating system's built-in permissions.

In addition to restricting the renderer's access to the filesystem and network, we can also place limitations on its access to the user's display and related objects. We run each render process on a separate Windows "Desktop" which is not visible to the user. This prevents a compromised renderer from opening new windows or capturing keystrokes.

Giving back memory

Given renderers running in separate processes, it becomes straightforward to treat hidden tabs as lower priority. Normally, minimized processes on Windows have their memory automatically put into a pool of "available memory." In low-memory situations, Windows will swap this memory to disk before it swaps out higher-priority memory, helping to keep the user-visible programs more responsive. We can apply this same principle to hidden tabs. When a render process has no top-level tabs, we can release that process's "working set" size as a hint to the system to swap that memory out to disk first if

necessary. Because we found that reducing the working set size also reduces tab switching performance when the user is switching between two tabs, we release this memory gradually. This means that if the user switches back to a recently used tab, that tab's memory is more likely to be paged in than less recently used tabs. Users with enough memory to run all their programs will not notice this process at all: Windows will only actually reclaim such data if it needs it, so there is no performance hit when there is ample memory.

This helps us get a more optimal memory footprint in low-memory situations. The memory associated with seldom-used background tabs can get entirely swapped out while foreground tabs' data can be entirely loaded into memory. In contrast, a single-process browser will have all tabs' data randomly distributed in its memory, and it is impossible to separate the used and unused data so cleanly, wasting both memory and performance.

Plug-ins

Firefox-style NPAPI plug-ins run in their own process, separate from renderers. This is described in detail in Plugin Architecture.

How to Add New Features (without bloating RenderView/RenderViewHost/WebContents)

Problem

Historically, new features (i.e. autofill to pick an example) have been added by bolting on their code to RenderView in the renderer process, and RenderViewHost in the browser process. If a feature was handled on the IO thread in the browser process, then its IPC messages were dispatched in BrowserMessageFilter. RenderViewHost would often dispatch the IPC message only to call WebContents (through the RenderViewHostDelegate interface), which would then call to another piece of code. All the IPC messages between the browser and renderer were declared in a massive render_messages_internal.h. Touching each of these files for every feature meant that the classes became bloated.

Solution

We have added helper classes and mechanisms for filtering IPC messages on each of the above threads. This makes it easier to write self contained features.

Renderer side:

If you want to filter and send IPC messages, implement the RenderViewObserver interface (content/renderer/render_view_observer.h). The RenderViewObserver base class takes a RenderView and manages the object's lifetime so that it's tied to RenderView (this is overridable). The class can then filter and send IPC messages, and additionally gets notification about frame loading and closing, which many features need. As an example, see ChromeExtensionHelper (chrome/renderer/extensions/chrome_extension_helper.h).

If your feature has part of the code in WebKit, avoid having callbacks go through WebViewClient interface so that we don't bloat it. Consider creating a new WebKit interface that the WebKit code calls, and have the renderer side class implement it. As an example, see WebAutoFillClient (WebKit/chromium/public/WebAutoFillClient.h).

Browser UI thread:

The WebContentsObserver (content/public/browser/web_contents_observer.h) interface allows objects on the UI thread to filter IPC messages and also gives notifications about frame navigation. As an example, see TabHelper (chrome/browser/extensions/tab_helper.h).

Browser other threads:

To filter and send IPC messages on other browser threads, such as IO/FILE/WEBKIT etc, implement BrowserMessageFilter interface (content/browser/browser_message_filter.h). The BrowserRenderProcessHost object creates and adds the filters in its CreateMessageFilters function.

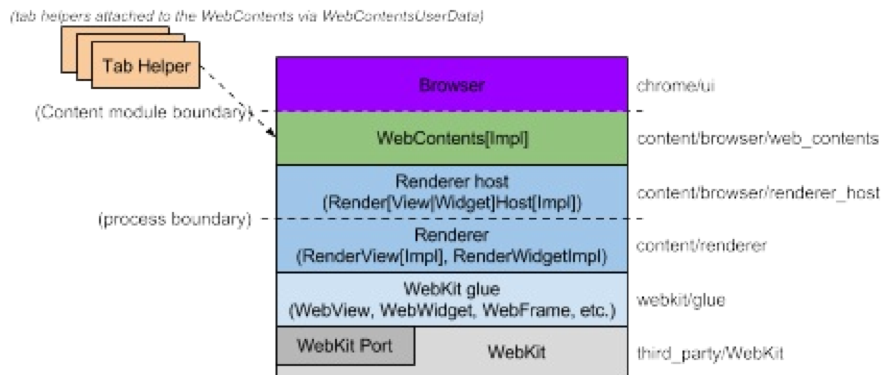
In general, if a feature has more than a few IPC messages, they should be moved into a separate file (i.e. not be added to render_messages_internal.h). This also helps with filtering messages on a thread other than the IO thread. As an example, see content/common/pepper_file_messages.h. This allows their filter, PepperFileMessageFilter, to easily send the messages to the file thread without having to specify their IDs in multiple places.

```
void PepperFileMessageFilter::OverrideThreadForMessage(
    const IPC::Message& message,
    BrowserThread::ID* thread) {
  if (IPC_MESSAGE_CLASS(message) == PepperFileMsgStart)
    *thread = BrowserThread::FILE;
}
```

How Chromium Displays Web Pages

This document describes how web pages are displayed in Chromium from the bottom up. Be sure you have read the multi-process architecture design document. You will especially want to understand the block diagram of major components. You may also be interested in multi-process resource loading for how pages are fetched from the network.

Conceptual application layers



(The original Google Doc for this illustration is http://goo.gl/MsEJX which is open for editing by any @chromium.org)

Each box represents a conceptual application layer. No layer should have knowledge of or dependencies on any higher-level layers.

- WebKit: Rendering engine shared between Safari, Chromium, and all other WebKit-based browsers. The Port is a part of WebKit that integrates with platform dependent system services such as resource loading and graphics.
- Glue: Converts WebKit types to Chromium types. This is our "WebKit embedding layer." It is the basis of two browsers, Chromium, and test_shell (which allows us to test WebKit).
- Renderer / Render host: This is Chromium's "multi-process embedding layer." It proxies notifications and commands across the process boundary.
- WebContents: A reusable component that is the main class of the Content module. It's easily embeddable to allow multiprocess rendering of HTML into a view. See the content module pages for more information.
- Browser: Represents the browser window, it contains multiple WebContentses.
- Tab Helpers: Individual objects that can be attached to a WebContents (via the WebContentsUserData mixin). The Browser attaches an assortment of them to the WebContentses that it holds (one for favicons, one for infobars, etc).

WebKit

We use the WebKit open-source project to lay out web pages. This code is pulled from Apple and stored in the /third_party/WebKit directory. WebKit consists primarily of "WebCore" which represents the core layout functionality, and "JavaScriptCore" which runs JavaScript. We only run JavaScriptCore for testing purposes, normally we replace it with our high performance V8 JavaScript engine. We do not actually use the layer that Apple calls "WebKit," which is the embedding API between WebCore and OS X applications such as Safari. We normally refer to the code from Apple generically as "WebKit" for convenience.

The WebKit port

At the lowest level we have our WebKit "port." This is our implementation of required platform-specific functionality that interfaces with the platform-independent WebCore code. These files are located in the WebKit tree, typically in chromium directories or as Chromium-suffixed files. Much of our port is not actually OS-specific: you could think of it as the "Chromium port" of WebCore. Some parts, like font rendering, must be handled differently for each platform.

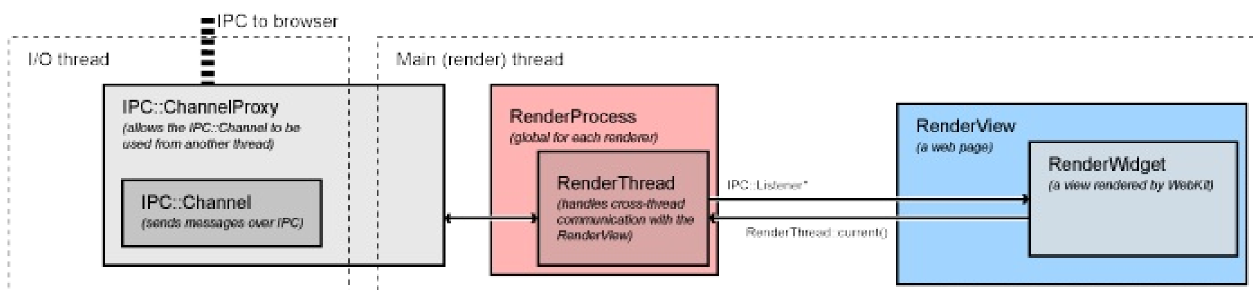
- Network traffic is handled by our multi-process resource loading system rather than being handed off to the OS directly from the render process.
- Graphics uses the Skia graphics library developed for Android. This is a cross-platform graphics library and handles all images and graphics primitives except for text. Skia is located in /third_party/skia. The main entrypoint for graphics operations is /webkit/port/platform/graphics/GraphicsContextSkia.cpp. It uses many other files in the same directory as well as from /base/gfx.

The WebKit glue

The Chromium application uses different types, coding styles, and code layout than the third-party WebKit code. The WebKit "glue" provides a more convenient embedding API for WebKit using Google coding conventions and types (for example, we use std::string instead of WebCore::String and GURL instead of KURL). The glue code is located in /webkit/glue. The glue objects are typically named similar to the WebKit objects, but with "Web" at the beginning. For example, WebCore::Frame becomes WebFrame. The WebKit "glue" layer insulates the rest of the Chromium code base from WebCore data types to help minimize the impact of WebCore changes on the Chromium code base. As such, WebCore data types are never used directly by Chromium. APIs are added to the WebKit "glue" for the benefit of Chromium when it needs to poke at some WebCore object.

The "test shell" application is a bare-bones web browser for testing our WebKit port and glue code. It uses the same glue interface for communicating with WebKit as Chromium does. It provides a simpler way for developers to test new code without having many complicated browser features, threads, and processes. This application is also used to run the automated WebKit tests. However, the downside of the "test shell" is that it doesn't exercise WebKit as Chromium does, in a multi-process way. The content module is embedded in an application called "content shell" which will soon be running the tests instead.

The render process



Chromium's render process embeds our WebKit port using the glue interface. It does not contain very much code: its job is primarily to be the renderer side of the IPC channel to the browser.. The most important class in the renderer is the RenderView, located in /content/renderer/render_view_impl.cc. This object represents a web page. It handles all navigation-related commands to and from the browser process. It derives from RenderWidget which provides painting and input event handling. The RenderView communicates with the browser process via the global (per render process) RenderProcess object.

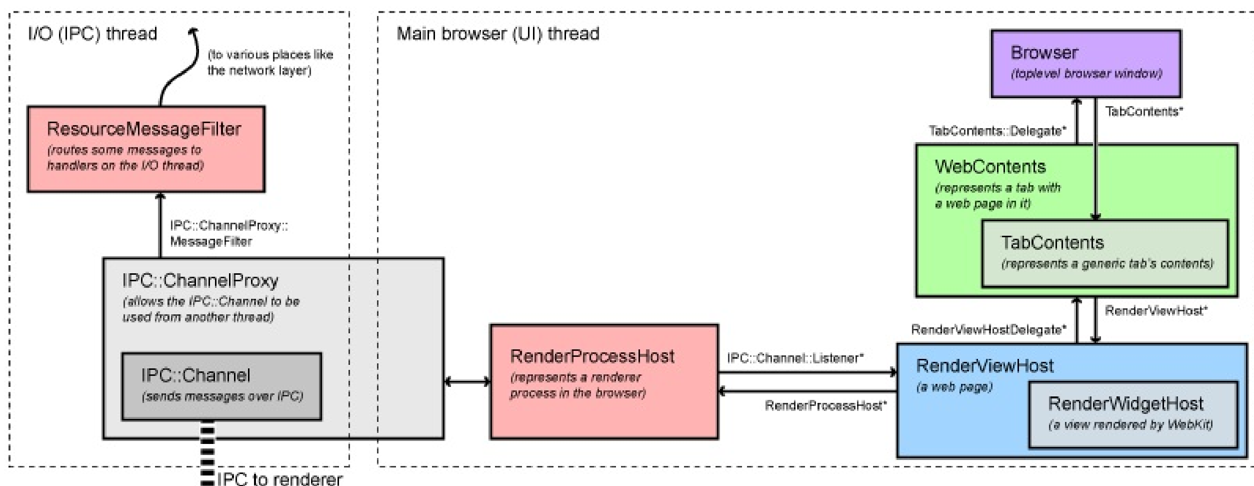
FAQ: What's the difference between RenderWidget and RenderView? RenderWidget maps to one WebCore::Widget object by implementing the abstract interface in the glue layer called WebWidgetDelegate.. This is basically a Window on the screen that receives input events and that we paint into. A RenderView inherits from RenderWidget and is the contents of a tab or popup Window. It handles navigational commands in addition to the painting and input events of the widget. There is only one case where a RenderWidget exists without a RenderView, and that's for select boxes on the web page.

These are the boxes with the down arrows that pop up a list of options. The select boxes must be rendered using a native window so that they can appear above everything else, and pop out of the frame if necessary. These windows need to paint and receive input, but there isn't a separate "web page" (RenderView) for them.

Threads in the renderer

Each renderer has two threads (see the multi-process architecture page for a diagram, or threading in Chromium in Chromium for how to program with them). The render thread is where the main objects such as the RenderView and all WebKit code run. When it communicates to the browser, messages are first sent to the main thread, which in turn dispatches the message to the browser process. Among other things, this allows us to send messages synchronously from the renderer to the browser. This happens for a small set of operations where a result from the browser is required to continue. An example is getting the cookies for a page when requested by JavaScript. The renderer thread will block, and the main thread will queue all messages that are received until the correct response is found. Any messages received in the meantime are subsequently posted to the renderer thread for normal processing.

The browser process



Low-level browser process objects

All IPC communication with the render processes is done on the I/O thread of the browser. This thread also handles all network communication which keeps it from interfering with the user interface.

When a RenderProcessHost is initialized on the main thread (where the user interface runs), it creates the new renderer process and a ChannelProxy IPC object with a named pipe to the renderer. This object runs on the I/O thread of the browser, listening to the named pipe to the renderer, and automatically forwards all messages back to the RenderProcessHost on the UI thread. A ResourceMessageFilter will be installed in this channel which will filter out certain messages that can be handled directly on the I/O thread such as network requests. This filtering happens in ResourceMessageFilter::OnMessageReceived.

The RenderProcessHost on the UI thread is responsible for dispatching all view-specific messages to the appropriate RenderViewHost (it handles a limited number of non-view-specific messages itself). This dispatching happens in RenderProcessHost::OnMessageReceived.

High-level browser process objects

View-specific messages come into RenderViewHost::OnMessageReceived. Most of the messages are handled here, and the rest get forwarded to the RenderWidgetHost base class. These two objects map to the RenderView and the RenderWidget in the renderer (see "The Render Process" above for what these mean). Each platform has a view class (RenderWidgetHostView[Aura|Gtk|Mac|Win]) to implement integration into the native view system.

Above the RenderView/Widget is the WebContents object, and most of the messages actually end up as function calls on that object. A WebContents represents the contents of a webpage. It is the top-level object in the content module, and has the responsibility of displaying a web page in a rectangular view. See the content module pages for more information.

The WebContents object is contained in a TabContentsWrapper. That is in chrome/ and is responsible for a tab.

Illustrative examples

Additional examples covering navigation and startup are in Getting Around the Chromium Source Code.

Life of a "set cursor" message

Setting the cursor is an example of a typical message that is sent from the renderer to the browser. In the renderer, here is what happens.

- Set cursor messages are generated by WebKit internally, typically in response to an input event. The set cursor message will start out in RenderWidget::SetCursor in content/renderer/render_widget.cc.
- It will call RenderWidget::Send to dispatch the message. This method is also used by RenderView to send messages to the browser. It will call RenderThread::Send.
- This will call the IPC::SyncChannel which will internally proxy the message to the main thread of the renderer and post it to the named pipe for sending to the browser.

Then the browser takes control:

- The IPC::ChannelProxy in the RenderProcessHost receives all message on the I/O thread of the browser. It first sends them through the ResourceMessageFilter that dispatches network requests and related messages directly on the I/O thread. Since our message is not filtered out, it continues on to the UI thread of the browser (the IPC::ChannelProxy does this internally).
- RenderProcessHost::OnMessageReceived in content/browser/renderer_host/render_process_host_impl.cc gets the messages for all views in the corresponding render process. It handles several types of messages directly, and for the rest forwards to the appropriate RenderViewHost corresponding to the source RenderView that sent the message.
- The message arrives at RenderViewHost::OnMessageReceived in content/browser/renderer_host/render_view_host_impl.cc. Many messages are handled here, but ours is not because it's a message sent from the RenderWidget and handled by the RenderWidgetHost.
- All unhandled messages in RenderViewHost are automatically forwarded to the RenderWidgetHost, including our set cursor message.
- The message map in content/browser/renderer_host/render_view_host_impl.cc finally receives the message in RenderWidgetHost::OnMsgSetCursor and calls the appropriate UI function to set the mouse cursor.

Life of a "mouse click" message

Sending a mouse click is a typical example of a message going from the browser to the renderer.

- The Windows message is received on the UI thread of the browser by RenderWidgetHostViewWin::OnMouseEvent which then calls ForwardMouseEventToRenderer in the same class.
- The forwarder function packages the input event into a cross-platform WebMouseEvent and ends up sending it to the RenderWidgetHost it is associated with.
- RenderWidgetHost::ForwardInputEvent creates an IPC message ViewMsg_HandleInputEvent, serializes the WebInputEvent to it, and calls RenderWidgetHost::Send.

- This just forwards to the owning RenderProcessHost::Send function, which in turn gives the message to the IPC::ChannelProxy.
- Internally, the IPC::ChannelProxy will proxy the message to the I/O thread of the browser and write it to the named pipe of the corresponding renderer.

Note that many other types of messages are created in the WebContents, especially navigational ones. These follow a similar path from the WebContents to the RenderViewHost.

Then the renderer takes control:

- IPC::Channel on the main thread of the renderer reads the message sent by the browser, and IPC::ChannelProxy proxies to the renderer thread.
- RenderView::OnMessageReceived gets the message. Many types messages are handled here directly. Since the click message is not, it falls through (with all other unhandled messages) to RenderWidget::OnMessageReceived which in turn forwards it to RenderWidget::OnHandleInputEvent.
- The input event is given to WebWidgetImpl::HandleInputEvent where it is converted to a WebKit PlatformMouseEvent class and given to the WebCore::Widget class inside WebKit.

See Also: Source code in design document

```
https://chromium.googlesource.com/chromium/src/+/master/docs/
```

General Architecture

- General Architecture
  - Conventions and patterns for multi-platform development
  - Extension Security Architecture: How the extension system helps reduce the severity of extension vulnerabilities
  - HW Video Acceleration in Chrom{e,ium}{,OS}
  - Inter-process Communication: How the browser, renderer, and plugin processes communicate
  - Multi-process Resource Loading: How pages and images are loaded from the network into the renderer
  - Plugin Architecture
  - Process Models: Our strategies for creating new renderer processes
  - Profile Architecture
  - SafeBrowsing
  - Sandbox
  - Security Architecture: How Chromium's sandboxed rendering engine helps protect against malware
  - Startup
  - Threading: How to use threads in Chromium Also see the documentation for V8, which is the JavaScript engine used within Chromium.

Conventions and patterns for multi-platform development

Chromium is a large and complex cross-platform product. We try to share as much code as possible between platforms, while implementing the UI and OS integration in the most appropriate way for each. While this gives a better user experience, it adds extra complexity to the code. This document describes the recommended practices for keeping such cross-platform code clean.

We use a variety of different file naming suffixes to indicate when a file should be used:

- Mac files use the _mac suffix for lower-level files and Cocoa (Mac UI) files use the _cocoa suffix.
- iOS files use the _ios suffix (although iOS also uses some specific _mac files).
- Linux files use _linux suffix for lower-level files, _gtk for GTK-specific files, and _x for X Windows (with no GTK) specific files.
- Windows files use the _win suffix.
- Posix files shared between Mac, iOS, and Linux use the _posix suffix.
- Files for Chrome's "Views" UI (on Windows and experimental GTK) layout system use the _views suffix.

The separate front-ends of the browser are contained in their own directories:

- Mac Cocoa: chrome/browser/ui/cocoa
- Linux GTK: chrome/browser/ui/gtk
- Windows Views (and the experimental GTK-views): chrome/browser/ui/views

The Coding Style page lists some stylistic rules affecting platform-specific defines.

How to separate platform-specific code

Small platform differences: #ifdefs

When you have a class with many shared functions or data members, but a few differences, use #ifdefs around the platform-specific parts. If there are no significant differences, it's easier for everybody to keep everything in one place.

Small platform differences in the header, larger ones in the implementation: split the implementation

There may be cases where there are few header file differences, but significant implementation differences for parts of the implementation. For example, base/waitable_event.h defines a common API with a couple of platform differences.

With significant implementation differences, the implementation files can be split. The prevents you from having to do a lot of #ifdefs for the includes necessary for each platform and also makes it easier to follow (three versions each of a set of functions in a file can get confusing). There can be different .cc files for each platform, as in base/waitable_event_posix.cc that implements posix-specific functions. If there were cross-platform functions in this class, they would be put in a file called base/waitable_event.cc.

Complete platform implementations and callers: separate implementations

When virtually none of the implementation is shared, implement the class separately for each platform in separate files.

If all implementations are in a cross-platform directory such as base, they should be named with the platform name, such as FooBarWin in base/foo_bar_win.h. This case will generally be rare since files in these cross-platform files are normally designed to be used by cross-platform code, and separate header files makes this impossible. In some places we've

defined a commonly named class in different files, so PlatformDevice is defined in skia/ext/platform_device_win.h, skia/ext/platform_device_linux.h, and skia/ext/platform_device_mac.h. This is OK if you really need to refer to this class in cross-platform code. But generally, cases like this will fall into the following rule.

If the implementations live in platform-specific directories such as chrome/browser/ui/cocoa or chrome/browser/ui/views, there is no chance that the class will be used by cross-platform code. In this case, the classes and filenames should omit the platform name since it would be redundant. So you would have FooBar implemented in chrome/browser/ui/cocoa/foo_bar.h.

Don't create different classes with different names for each platform and typedef it to a shared name. We used to have this for PlatformCanvas, where it was a typedef of PlatformCanvasMac, PlatformCanvasLinux, or PlatformCanvasWin depending on the platform. This makes it impossible to forward-declare the class, which is an important tool for reducing dependencies.

When to use virtual interfaces

In general, virtual interfaces and factories should not be used for the sole purpose of separating platform differences. Instead, it should be be used to separate interfaces from implementations to make the code better designed. This comes up mostly when implementing the view as separate from the model, as in TabContentsView or RenderWidgetHostView. In these cases, it's desirable for the model not to depend on implementation details of the view. In many cases, there will only be one implementation of the view for each platform, but gives cleaner separation and more flexibility in the future.

In some places like TabContentsView, the virtual interface has non-virtual functions that do things shared between platforms. Avoid this. If the code is always the same regardless of the view, it probably shouldn't be in the view in the first place.

Implementing platform-specific UI

In general, construct platform specific user interface elements from other platform-specific user interface elements. For instance, the views-specific class BrowserView is responsible for constructing many of the browser dialog boxes. The alternative is to wrap the UI element in a platform-independent interface and construct it from a model via a factory. This is significantly less desirable as it confuses ownership: in most cases of construction by factory, the UI element returned ends up being owned by the model that created it. However in many cases the UI element is most easily managed by the UI framework to which it belongs. For example, a views::View is owned by its view hierarchy and is automatically deleted when its containing window is destroyed. If you have a dialog box views::View that implements a platform independent interface that is then owned by another object, the views::View instance now needs to explicitly tell its view hierarchy not to try and manage its lifetime.

e.g. prefer this:

```
// browser.cc:

Browser::ExecuteCommand(..) {
  ...
  case IDC_COMMAND_EDIT_FOO:
    window()->ShowFooDialog();
    break;
  ...
}

// browser_window.h:

class BrowserWindow {
...
  virtual void ShowFooDialog() = 0;
...
};

// browser_view.cc:

BrowserView::ShowFooDialog() {
  views::Widget::CreateWindow(new FooDialogView)->Show();
}

// foo_dialog_view.cc:

// FooDialogView and FooDialogController are automatically cleaned up when the window is closed.
class FooDialogView : public views::View {
  ...
 private:
  scoped_ptr<FooDialogController> controller_; // Cross-platform state/control logic
  ...
}
```

to this:

```
// browser.cc:

Browser::ExecuteCommand(..) {
  ...
  case IDC_COMMAND_EDIT_FOO: {
    FooDialogController::instance()->ShowUI();
    break;
  }
  ...
}

// foo_dialog_controller.h:

class FooDialog {
 public:
  static FooDialog* CreateFooDialog(FooDialogController* controller);
  virtual void Show() = 0;
  virtual void Bar() = 0;
};

class FooDialogController {
 public:
  ...
  static FooDialogController* instance() {
    static FooDialogController* instance = NULL;
    if (!instance)
      instance = Singleton<FooDialogController>::get();
    return instance;
  }
  ...
 private:
  ...
  void ShowUI() {
    if (!dialog_.get())
      dialog_.reset(FooDialog::CreateFooDialog(this));
    dialog_->Show();
  }

  // Why bother keeping FooDialog or even FooDialogController around?
  // Most dialogs are very seldom used.
  scoped_ptr<FooDialog> dialog_;
};

// foo_dialog_win.cc:

class FooDialogView : public views::View,
                      public FooDialogController {
 public:
  ...
  explicit FooDialogView(FooDialogController* controller) {
    set_parent_owned(false);  // Now necessary due to scoped_ptr in FooDialogController.
  }
  ...
};

FooDialog* FooDialog::CreateFooDialog(FooDialogController* controller) {
  return new FooDialogView(controller);
}
```

Sometimes this latter pattern is necessary, but these occasions are rare, and very well understood by the frontend team. When porting, consider converting cases of the latter model to the former model if the UI element is something simple like a dialog box.

Web Security Research

Protecting Browsers from Extension Vulnerabilities

Protecting Browsers from Extension Vulnerabilities

Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman

EECS Department. University of California, Berkeley. Technical Report No. UCB/EECS-2009-185

Abstract

Browser extensions are remarkably popular, with one in three Firefox users running at least one extension. Although well-intentioned, extension developers are often not security experts and write buggy code that can be exploited by malicious web site operators. In the Firefox extension system, these exploits are dangerous because extensions run with the user's full privileges and can read and write arbitrary files and launch new processes. In this paper, we analyze 25 popular Firefox extensions and find that 88% of these extensions need less than the full set of available privileges. Additionally, we find that 76% of these extensions use unnecessarily powerful APIs, making it difficult to reduce their privileges. We propose a new browser extension system that improves security by using least privilege, privilege separation, and strong isolation. Our system limits the misdeeds an attacker can perform through an extension vulnerability. Our design has been adopted as the Google Chrome extension system.

An extended version of this paper will appear at Proc. of the 17th Network and Distributed System Security Symposium (NDSS 2010).

More Berkeley web security research >>

HW Video Acceleration in Chrom{e,ium}{,OS}

Ami Fischman fischman@chromium.org

Status as of 2014/06/06: Up-to-date

(could use some more details)

Introduction

Video decode (e.g. YouTube playback) and encode (e.g. video chat applications) are some of the most complex compute operations on the modern web. Moving these operations from software running on general-purpose CPUs to dedicated hardware blocks means lower power consumption, longer battery life, higher quality (e.g. HD instead of SD), and better interactive performance as the CPU is freed up to work on everything else it needs to do.

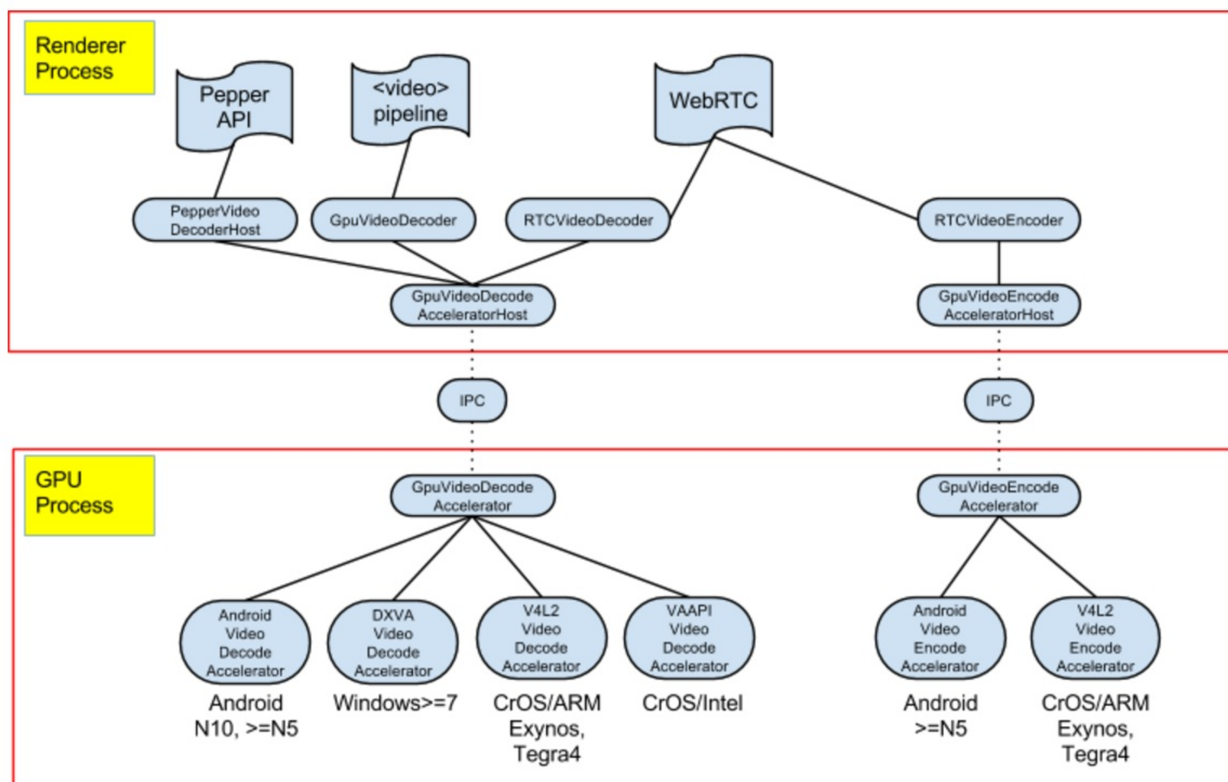
Design

media::VideoDecodeAccelerator (VDA) and media::VideoEncodeAccelerator (VEA) (with their respective Client subclasses) are the interfaces at the center of all video HW acceleration in Chrome. Each consumer of HW acceleration implements the relevant Client interface and calls an object of the relevant V[DE]A interface.

In general the classes that want to encode or decode video live in the renderer process (e.g. the player, or WebRTC's video encoders & decoders) and the HW being utilized is not accessible from within the renderer process, so IPC is used to bridge the renderer<->GPU process gap.

Implementation Details

The main consumers of the acceleration APIs are:
pipeline (what plays media on the web), WebRTC (enabling plugin-free real-time video chat on the web), and Pepper API (offering HW acceleration to pepper plugins such as Adobe Flash). The implementations of the acceleration APIs are specific to the OS (and sometimes HW platform) due to radically different options offered by the OS and drivers/HW present.



(not pictured: obsolete OpenMAX-IL-based OVDA, and never-launched MacVDA).

Current Status

New devices are released all the time so this list is likely already out of date, but as of early June 2014, existing (public) support includes: **Decode**

- Windows: starting with Windows 7, HW accelerated decode of h.264 is used via DXVAVDA.
- CrOS/Intel (everything post-Mario/Alex/ZGB): HW accelerated decode of h.264 is used via VAVDA
- CrOS/ARM: HW accelerated decode of VP8 and h.264 is available via V4L2VDA
- Android: HW accelerated decode of VP8 is available on N10, N5, some S4's, and a bunch of other devices. (note that on Android this only applies to WebRTC, as there is no PPAPI and uses the platform's player) **Encode**
- CrOS/ARM: HW accelerated encode of h.264 (everywhere) and VP8 (2014 devices) is available via V4L2VEA
- Android: HW accelerated encode of VP8 is available on N5.

Results

Generally speaking offloading encode or decode from CPU to specialized HW has shown an overall battery-life extension of 10-25% depending on the platform, workload, etc. For some data examples see: public: 133827#c27, 219957#c16 google-internal only: summary slide deck, CrOS/ARM-1, CrOS/ARM-2

Inter-process Communication (IPC)

Overview

Chromium has a multi-process architecture which means that we have a lot of processes communicating with each other. Our main inter-process communication primitive is the named pipe. On Linux & OS X, we use a socketpair(). A named pipe is allocated for each renderer process for communication with the browser process. The pipes are used in asynchronous mode to ensure that neither end is blocked waiting for the other.

For advice on how to write safe IPC endpoints, please see Security Tips for IPC.

IPC in the browser

Within the browser, communication with the renderers is done in a separate I/O thread. Messages to and from the views then have to be proxied over to the main thread using a ChannelProxy. The advantage of this scheme is that resource requests (for web pages, etc.), which are the most common and performance critical messages, can be handled entirely on the I/O thread and not block the user interface. These are done through the use of a ChannelProxy::MessageFilter which is inserted into the channel by the RenderProcessHost. This filter runs in the I/O thread, intercepts resource request messages, and forwards them directly to the resource dispatcher host. See Multi-process Resource Loading for more information on resource loading.

IPC in the renderer

Each renderer also has a thread that manages communication (in this case, the main thread), with the rendering and most processing happening on another thread (see the diagram in multi-process architecture). Most messages are sent from the browser to the WebKit thread through the main renderer thread and vice-versa. This extra thread is to support synchronous renderer-to-browser messages (see "Synchronous messages" below).

Messages

Types of messages

We have two primary types of messages: "routed" and "control." Control messages are handled by the class that created the pipe. Sometimes that class will allow others to received message by having a MessageRouter object that other listeners can register with and received "routed" messages sent with their unique (per pipe) id.

For example, when rendering, control messages are not specific to a given view and will be handled by the RenderProcess (renderer) or the RenderProcessHost (browser). Requests for resources or to modify the clipboard are not view-specific so are control messages. An example of routed messages are a message to ask a view to paint a region.

Routed messages have historically been used to get messages to a specific RenderViewHost. However, technically any class can receive routed messages by using RenderProcessHost::GetNextRoutingID and registering itself with RenderProcessHost::AddRoute. Currently both RenderFrameHost and RenderViewHost have their own routing IDs.

Independent of the message type is whether the message is sent from the browser to the renderer, or from the renderer to the browser. Messages sent from the browser to the renderer are called View messages because they are being sent to the RenderView. Messages sent from the renderer to the browser are called ViewHost messages because they are being sent to the RenderViewHost. You will notice the messages defined in render_messages_internal.h are separated into these two categories.

Plugins also have separate processes. Like the render messages, there are PluginProcess messages (sent from the browser to the plugin process) and PluginProcessHost messages (sent from the plugin process to the browser). These messages are all defined in plugin_messages_internal.h. The automation messages (for controlling the browser from the UI tests) are done in a similar manner.

Declaring messages

Special macros are used to declare messages. The messages sent between the renderer and the browser are all declared in render_messages_internal.h. There are two sections, one for "View" messages sent to the renderer, and one for "ViewHost" messages sent to the browser.

To declare a message from the renderer to the browser (a "ViewHost" message) that is specific to a view ("routed") that contains a URL and an integer as an argument, write:

IPC_MESSAGE_ROUTED2(ViewHostMsg_MyMessage, GURL, int)

To declare a control message from the browser to the renderer (a "View" message) that is not specific to a view ("control") that contains no parameters, write:

IPC_MESSAGE_CONTROL0(ViewMsg_MyMessage)

Pickling values

Parameters are serialized and de-serialized to message bodies using the ParamTraits template. Specializations of this template are provided for most common types in ipc_message_utils.h. If you define your own types, you will also have to define your own ParamTraits specialization for it.

Sometimes, a message has too many values to be reasonably put in a message. In this case, we define a separate structure to hold the values. For example, for the FrameMsg_Navigate message, the CommonNavigationParams structure is defined in navigation_params.h. frame_messages.h defines the ParamTraits specializations for the structures using the IPC_STRUCT_TRAITS family of macros.

Sending messages

You send messages through "channels" (see below). In the browser, the RenderProcessHost contains the channel used to send messages from the UI thread of the browser to the renderer. The RenderWidgetHost (base class for RenderViewHost) provides a Send function that is used for convenience.

Messages are sent by pointer and will be deleted by the IPC layer after they are dispatched. Therefore, once you can find the appropriate Send function, just call it with a new message:

Send(new ViewMsg*StopFinding(routing_id*)); Notice that you must specify the routing ID in order for the message to be routed to the correct View/ViewHost on the receiving end. Both the RenderWidgetHost (base class for RenderViewHost) and the RenderWidget (base class for RenderView) have GetRoutingID() members that you can use.

Handling messages

Messages are handled by implementing the IPC::Listener interface, the most important function on which is OnMessageReceived. We have a variety of macros to simplify message handling in this function, which can best be illustrated by example:

```
MyClass::OnMessageReceived(const IPC::Message& message) {
  IPC_BEGIN_MESSAGE_MAP(MyClass, message)
    // Will call OnMyMessage with the message. The parameters of the message will be unpacked for you.
    IPC_MESSAGE_HANDLER(ViewHostMsg_MyMessage, OnMyMessage)
    ...
    IPC_MESSAGE_UNHANDLED_ERROR()  // This will throw an exception for unhandled messages.
  IPC_END_MESSAGE_MAP()
}

// This function will be called with the parameters extracted from the ViewHostMsg_MyMessage message.
MyClass::OnMyMessage(const GURL& url, int something) {
  ...
}
```

You can also use IPC_DEFINE_MESSAGE_MAP to implement the function definition for you as well. In this case, do not specify a message variable name, it will declare a OnMessageReceived function on the given class and implement its guts.

Other macros:

- IPC_MESSAGE_FORWARD: This is the same as IPC_MESSAGE_HANDLER but you can specify your own class to send the message to, instead of sending it to the current class. IPC_MESSAGE_FORWARD(ViewHostMsg_MyMessage, some_object_pointer, SomeObject::OnMyMessage)
- IPC_MESSAGE_HANDLER_GENERIC: This allows you to write your own code, but you have to unpack the parameters from the message yourself:

IPC_MESSAGE_HANDLER_GENERIC(ViewHostMsg_MyMessage, printf("Hello, world, I got the message."))

Security considerations

Security bugs in IPC can have nasty consequences (file theft, sandbox escapes, remote code execution). Check out our security for IPC document for tips on how to avoid common pitfalls.

Channels

IPC::Channel (defined in ipc/ipc_channel.h) defines the methods for communicating across pipes. IPC::SyncChannel provides additional capabilities for synchronously waiting for responses to some messages (the renderer processes use this as described below in the "Synchronous messages" section, but the browser process never does).

Channels are not thread safe. We often want to send messages using a channel on another thread. For example, when the UI thread wants to send a message, it must go through the I/O thread. For this, we use a IPC::ChannelProxy. It has a similar API as the regular channel object, but proxies messages to another thread for sending them, and proxies messages back to the original thread when receiving them. It allows your object (typically on the UI thread) to install a IPC::ChannelProxy::Listener on the channel thread (typically the I/O thread) to filter out some messages from getting proxied over. We use this for resource requests and other requests that can be handled directly on the I/O thread. RenderProcessHost installs a RenderMessageFilter object that does this filtering.

Synchronous messages

Some messages should be synchronous from the renderer's perspective. This happens mostly when there is a WebKit call to us that is supposed to return something, but that we must do in the browser. Examples of this type of messages are spell-checking and getting the cookies for JavaScript. Synchronous browser-to-renderer IPC is disallowed to prevent blocking the user-interface on a potentially flaky renderer.

Danger: Do not handle any synchronous messages in the UI thread! You must handle them only in the I/O thread. Otherwise, the application might deadlock because plug-ins require synchronous painting from the UI thread, and these will be blocked when the renderer is waiting for synchronous messages from the browser.

Declaring synchronous messages

Synchronous messages are declared using the IPC*SYNC_MESSAGE** macros. These macros have input and return parameters (non-synchronous messages lack the concept of return parameters). For a control function which takes two input parameters and returns one parameter, you would append 2_1 to the macro name to get:

IPC_SYNC_MESSAGE_CONTROL2_1(SomeMessage, // Message name GURL, //input_param1 int, //input_param2 std::string); //result Likewise, you can also have messages that are routed to the view in which case you would replace "control" with "routed" to get IPC_SYNC_MESSAGE_ROUTED2_1. You can also have 0 input or return parameters. Having no return parameters is used when the renderer must wait for the browser to do something, but needs no results. We use this for certain printing and clipboard operations.

Issuing synchronous messages

When the WebKit thread issues a synchronous IPC request, the request object (derived from IPC::SyncMessage) is dispatched to the main thread on the renderer through a IPC::SyncChannel object (the same one is also used to send all asynchronous messages). The SyncChannel will block the calling thread when it receives a synchronous message, and will only unblock it when the reply is received.

While the WebKit thread is waiting for the synchronous reply, the main thread is still receiving messages from the browser process. These messages will be added to the queue of the WebKit thread for processing when it wakes up. When the synchronous message reply is received, the thread will be un-blocked. Note that this means that the synchronous message reply can be processed out-of-order.

Synchronous messages are sent the same way normal messages are, with output parameters being given to the constructor. For example:

```
const GURL input_param("http://www.google.com/");
std::string result;
content::RenderThread::Get()->Send(new MyMessage(input_param, &result));
printf("The result is %s\n", result.c_str());
```

Handling synchronous messages

Synchronous messages and asynchronous messages use the same IPC_MESSAGE_HANDLER, etc. macros for dispatching the message. The handler function for the message will have the same signature as the message constructor, and the function will simply write the output to the output parameter. For the above message you would add

IPC_MESSAGE_HANDLER(MyMessage, OnMyMessage) to the OnMessageReceived function, and write:

```
void RenderProcessHost::OnMyMessage(GURL input_param, std::string* result) {
  *result = input_param.spec() + " is not available";
}
```

Converting message type to a message name

If you get a crash and you have the message type you can convert this to a message name. The message type will be 32-bit value, the high 16-bits are the class and the low 16-bits are the id. The class is based on the enums in ipc/ipc_message_start.h, the id is based on the line number in the file that defines the message. This means that you need to get the exact revision of Chromium in order to accurately get the message name.

Example of this in 554011 was 0x1c0098 at Chromium revision ad0950c1ac32ef02b0b0133ebac2a0fa4771cf20. That's class 0x1c which is line 40 which matches ChildProcessMsgStart. ChildProcessMsgStart messages are in content/common/child_process_messages.h and the IPC will be on line 0x98 or line 152 which is ChildProcessHostMsg_ChildHistogramData.

This technique is particularly useful if you are dealing with crashes caused by content::RenderProcessHostImpl::OnBadMessageReceived

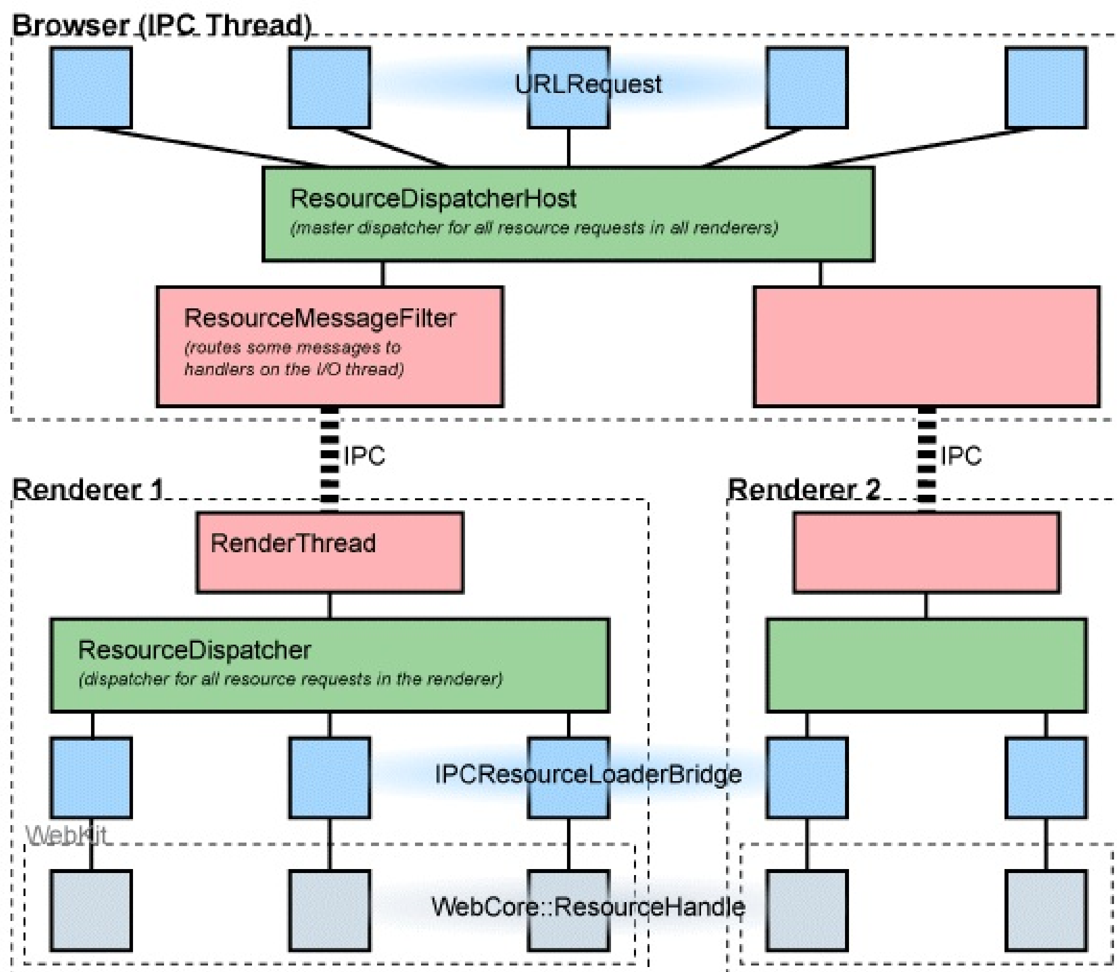
Multi-process Resource Loading（need update）

Background

All network communication is handled by the main browser process. This is done not only so that the browser process can control each renderer's access to the network, but also so that we can maintain consistent session state across processes like cookies and cached data. It is also important because as a HTTP/1.1 user-agent, the browser as a whole should not open too many connections per host.

Overview

Our multi-process application can be viewed in three layers. At the lowest layer is the WebKit library which renders pages. Above that are the renderer process (simplistically, one-per-tab), each of which contains one WebKit instance. Managing all the renderers is the browser process, which controls all network accesses.



Blink

Blink has a ResourceLoader object which is responsible for fetching data. Each loader has a WebURLLoader for performing the actual requests. The header file for this interface is inside the Blink repo.

ResourceLoader implements the interface WebURLLoaderClient. This is the callback interface used by the renderer to dispatch data and other events to Blink.

The test shell uses a different resource loader, so provides a different implementation, non-IPC version of ResourceLoaderBridge, located in webkit/tools/test_shell/simple_resource_loader_bridge.

Renderer

The renderer's implementation of WebURLLoader, called WebURLLoaderImpl, is located in content/child/. It uses the global ResourceDispatcher singleton object (one for each renderer process) to create a unique request ID and forward the request to the browser via IPC. Responses from the browser will reference this request ID, which can then be converted back to the RequestPeer object (WebURLRequestImpl) by the resource dispatcher.

Browser

The RenderProcessHost objects inside the browser receive the IPC requests from each renderer. It forwards these requests to the global ResourceDispatcherHost, using a pointer to the render process host (specifically, an implementation of ResourceDispatcherHost::Receiver) and the request ID generated by the renderer to uniquely identify the request.

Each request is then converted into a URLRequest object, which in turn forwards it to it's internal URLRequestJob that implements the specific protocol desired. When the URLRequest generates notifications, its ResourceDispatcherHost::Receiver and request ID are used to send the notification to the correct RenderProcessHost for sending back to the renderer. Since the ID generated by the renderer is preserved, it is able to correlate all responses with a specific request first generated by WebKit.

Cookies

All cookies are handled by our CookieMonster object in /net/base. We do not share cookies with WinInet. The cookie monster lives in the browser process which handles all network requests because cookies need to be the same across all tabs.

Pages can request cookies for a document via document.cookie. When this occurs, we send a synchronous message from the renderer to the browser requesting the cookie. While the browser is processing the cookie, the thread that WebKit works on is suspended. When the renderer's I/O thread receives the response from the browser, it un-suspends the thread and passes the result back to the JavaScript engine.

Plugin Architecture

Background

Before reading this document, you should be familiar with Chromium's multi-process architecture.

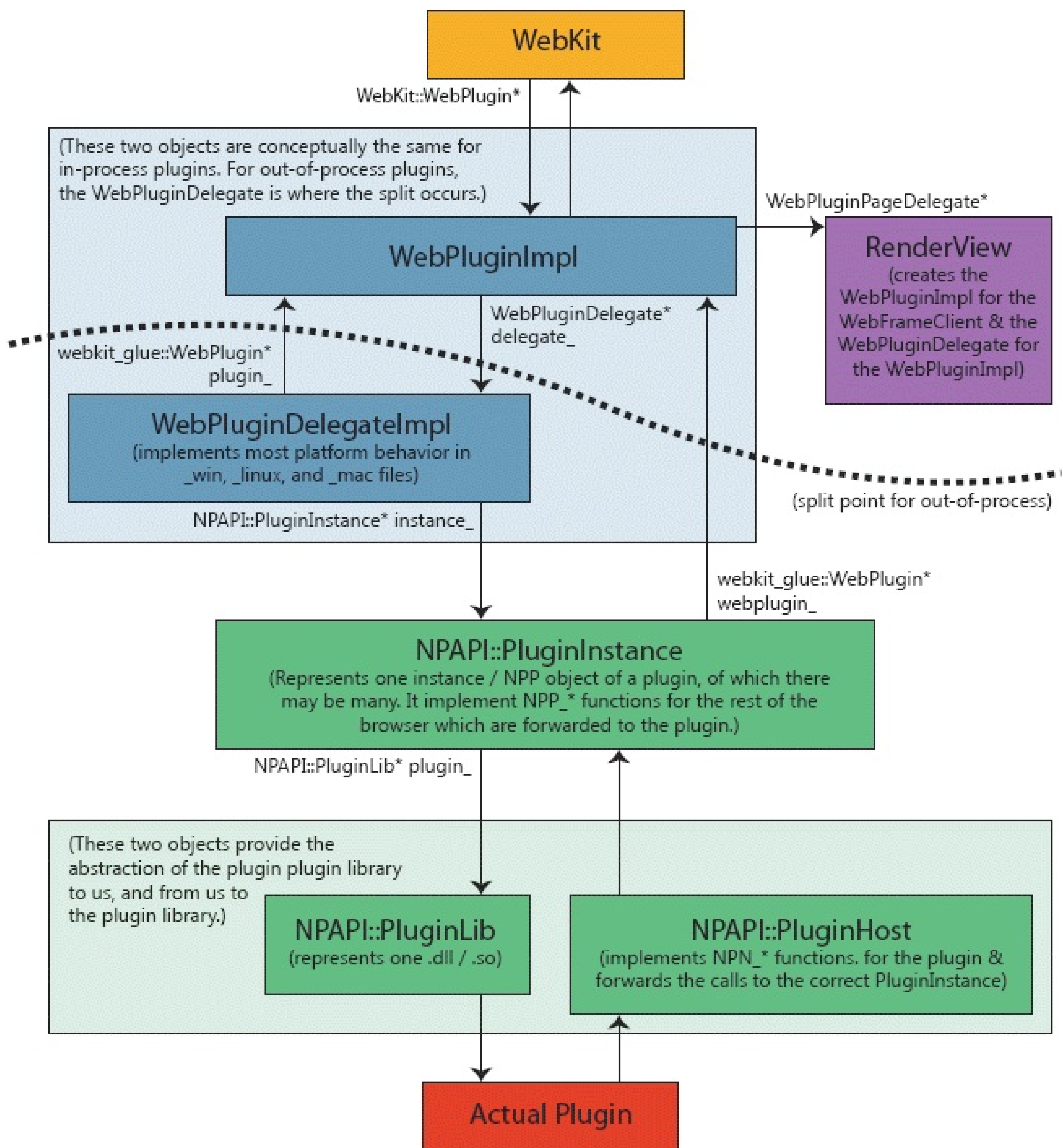
Overview

Plugins are a major source of browser instability. Plugins also make sandboxing the process where the renderer runs impractical, as plugins are written by third-parties and we can't control their access to the operating system. The solution is to run plugins in their own separate process.

Detailed design

In-process plugins

Chromium has the ability to run plugins in process (this is handy for testing) as well as out of process. Both start at our non-multi-process-aware WebKit embedding layer, which expects the embedder to implement the WebKit::WebPlugin interface. This is implemented by WebPluginImpl. The WebPluginImpl talks "up" the chain to a WebPluginDelegate interface, which for in-process plugins is implemented by WebPluginDelegateImpl. This in turn talks to our NPAPI wrapper layer.



Historical note: Before we had the WebKit embedding layer, WebPluginImpl was the embedding layer. It would talk to the "embedding application" through the WebPluginDelegate abstract interface, which we would switch implementations of for in-process and out-of-process plugins. With the addition of the Chromium WebKit API, we added the new WebKit::WebPlugin abstract interface, which has the same function as the old WebPluginDelegate interface. A better design with this interface would be to merge WebPluginImpl and WebPluginDelegateImpl and do the process split at the WebKit::WebPlugin level. This has not been changed due to its complexity.

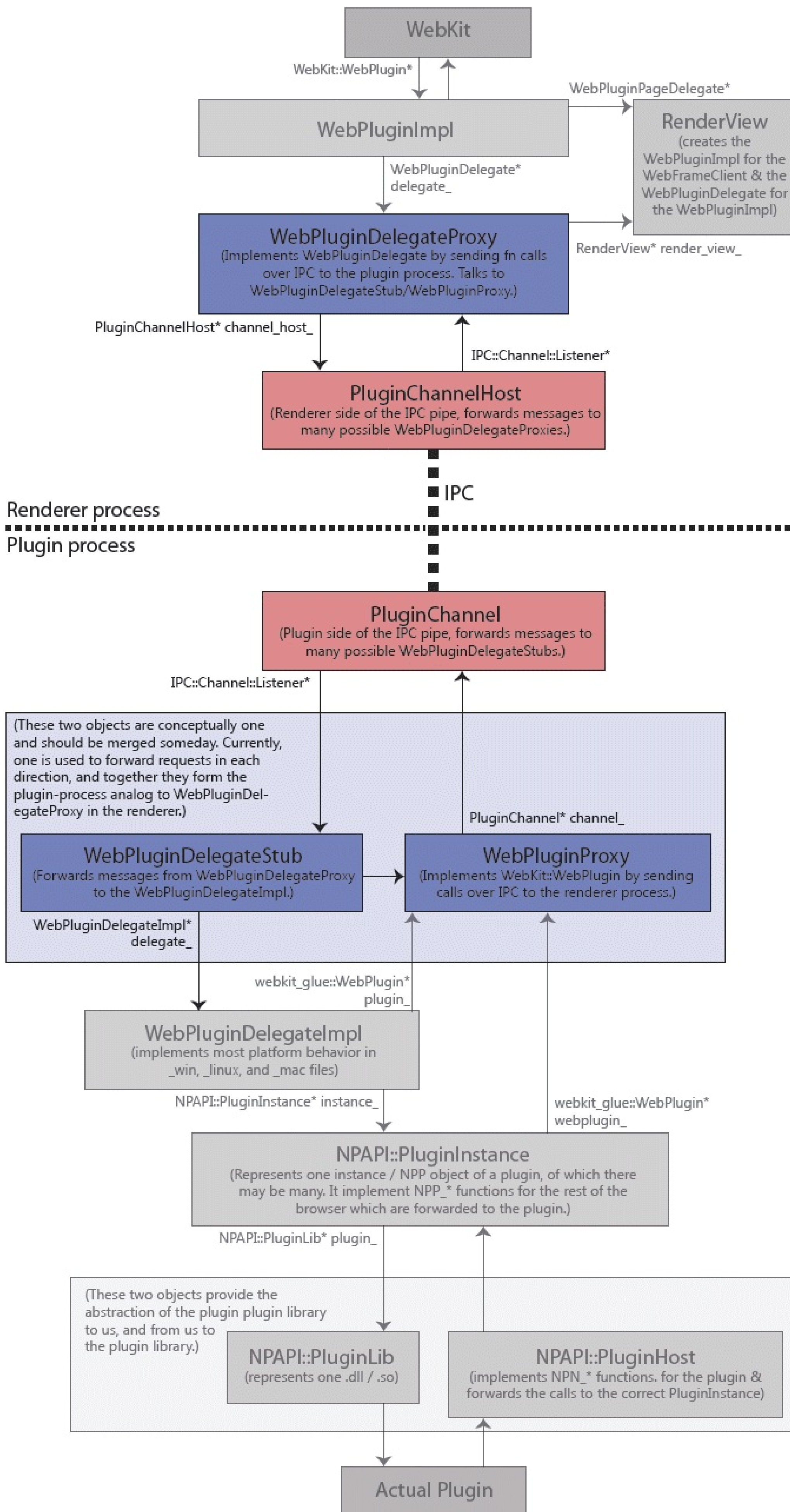
Out-of-process plugins

Chromium switches out implementations at the layer indicated by the dotted line in the diagram above to support out-of-process plugins. This just interposes an IPC layer between the WebPluginImpl and WebPluginDelegateImpl layers, and lets us share all of our NPAPI code between each mode. All of the old WebPluginDelegateImpl code, as well as all of the NPAPI layer it talks to, now executes in the separate plugin process.

The two sides of a Renderer/Plugin communication channel are represented by the PluginChannel and the PluginChannelHost. We have many renderer processes, and one plugin process for each unique plugin. This means there is one PluginChannelHost in the renderer for each type of plugin it uses (for example, Adobe Flash and Windows Media Player). In each plugin process, there will be one PluginChannel for each renderer process that has an instance of that type of plugin.

Each end of the channel, in turn, maps to many different instances of a plugin. For example, if there are two Adobe Flash movies embedded in a web page, there will be two WebPluginDelegateProxies (and related stuff) in the renderer side, and two WebPluginDelegateStubs (and related stuff) in the plugin side. The channel is in charge of multiplexing the communication between these many objects over one IPC connection.

In this diagram, you can see the classes from the above in-process diagram grayed out, with the new out-of-process layer (in color) in the middle.


WebKit
WebKit::WebPlugin*
WebPluginPageDelegate*
WebPluginImpl
RenderView
(creates the WebPluginImpl for the WebFrameClient & the WebPluginDelegate for the WebPluginImpl)
WebPluginDelegate* delegate_
WebPluginDelegateProxy
(Implements WebPluginDelegate by sending fn calls over IPC to the plugin process. Talks to WebPluginDelegateStub/WebPluginProxy.)
RenderView* render_view_
PluginChannelHost* channel_host_
IPC::Channel::Listener*
PluginChannelHost
(Renderer side of the IPC pipe, forwards messages to many possible WebPluginDelegateProxies.)
IPC
Renderer process
Plugin process
PluginChannel
(Plugin side of the IPC pipe, forwards messages to many possible WebPluginDelegateStubs.)
IPC::Channel::Listener*
(These two objects are conceptually one and should be merged someday. Currently, one is used to forward requests in each direction, and together they form the plugin-process analog to WebPluginDel-egateProxy in the renderer.)
PluginChannel* channel_
WebPluginDelegateStub
(Forwards messages from WebPluginDelegateProxy to the WebPluginDelegateImpl.)
WebPluginProxy
(Implements WebKit::WebPlugin by sending calls over IPC to the renderer process.)
WebPluginDelegateImpl* delegate_
webkit_glue::WebPlugin* plugin_
WebPluginDelegateImpl
(implements most platform behavior in _win, _linux, and _mac files)
NPAPI::PluginInstance* instance_
webkit_glue::WebPlugin* webplugin_
NPAPI::PluginInstance
(Represents one instance / NPP object of a plugin, of which there may be many. It implement NPP_* functions for the rest of the browser which are forwarded to the plugin.)
NPAPI::PluginLib* plugin_
(These two objects provide the abstraction of the plugin plugin library to us, and from us to the plugin library.)
NPAPI::PluginLib
(represents one .dll / .so)
NPAPI::PluginHost
(implements NPN_* functions. for the plugin & forwards the calls to the correct PluginInstance)
Actual Plugin

Historical note: We used to consistently use a stub/proxy model for communication, with one stub and one proxy on each side of the IPC channel for receiving and sending messages to one plugin, respectively. This leads to many classes that can get confusing. As a result, the WebPluginStub was merged into the WebPluginDelegateProxy which now handles the renderer side of all IPC communication for one plugin instance. The plugin side has not been merged yet, leaving two classes, the WebPluginDelegateStub and the WebPluginProxy as conceptually the same object, just representing different directions of communication.

Windowless Plugins

Windowless plugins are designed to run directly within the rendering pipeline. When WebKit wants to draw a region of the screen involving the plugin it calls into the plugin code, handing it a drawing context. Windowless plugins are often used in situations where the plugin is expected to be transparent over the page -- it's up to the plugin drawing code to decide how it munges the bit of the page it's given.

To take windowless plugins out of process, you still need to incorporate their rendering in the (synchronous) rendering pass done by WebKit. A naively slow option is to clip out the region that the plugin will draw on then synchronously ship that over to the plugin process and let it draw. This can then be sped up with some shared memory.

However, rendering speed is then at the mercy of the plugin process (imagine a page with 30 transparent plugins -- we'd need 30 round trips to the plugin process). So instead we have windowless plugins asynchronously paint, much like how our existing page rendering is asynchronous with respect to the screen. The renderer has effectively a backing store of what the plugin's rendered area looks like and uses this image when drawing, and the plugin is free to asynchronously send over new updates representing changes to the rendered area.

All of this is complicated a bit by "transparent" plugins. The plugin process needs to know what pixels it wants to draw over. So it also keeps a cache of what the renderer last sent it as the page background behind the plugin, then lets the plugin repeatedly draw over that.

So, in all, here are the buffers involved for the region drawn by a windowless plugin:

Renderer process

- backing store of what the plugin last rendered as
- shared memory with the plugin for receiving updates ("transport DIB")
- copy of the page background behind the plugin (described below)

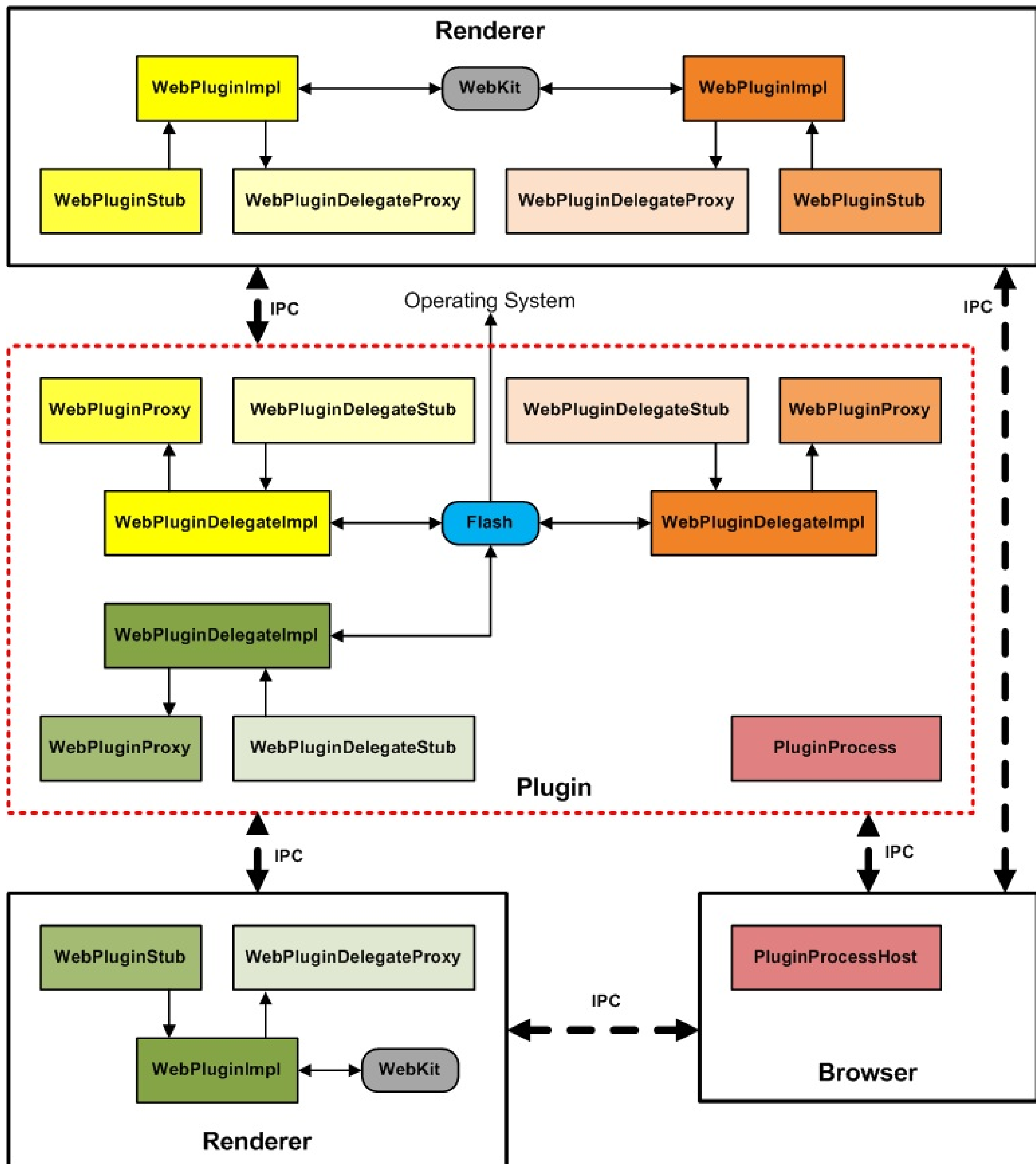
Plugin process

- copy of the page background behind the plugin, used as the source material when drawing
- shared memory with the renderer for sending updates ("transport DIB")

Why does the renderer keep a copy of the page background? Because if the page background changes, we need to synchronously get the plugin to redraw over the new background it will appear to lag. We can tell that the background changed by comparing the newly-rendered background against our copy of what the plugin thinks the background. Since the plugin and renderer processes are asynchronous with respect to one another, they need separate copies.

Overall system

This image shows the overall system with the browser and two renderer processes, each communicating with one shared out-of-process Flash process. There are three total plugin instances. Note that this diagram is out of date, and WebPluginStub has been merged with WebPluginDelegateProxy.


Renderer
WebPluginImpl
WebKit
WebPluginImpl
WebPluginStub
WebPluginDelegateProxy
WebPluginDelegateProxy
WebPluginStub
IPC
Operating System
IPC
WebPluginProxy
WebPluginDelegateStub
WebPluginDelegateStub
WebPluginProxy
WebPluginDelegateImpl
Flash
WebPluginDelegateImpl
WebPluginDelegateImpl
WebPluginProxy
WebPluginDelegateStub
PluginProcess
Plugin
IPC
IPC
WebPluginStub
WebPluginDelegateProxy
PluginProcessHost
IPC
WebPluginImpl
WebKit
Browser
Renderer

Process Models

This document describes the different process models that Chromium supports for its renderer processes, as well as caveats in the models as it exists currently.

Overview

Web content has evolved to contain significant amounts of active code that run within the browser, making many web sites more like applications than documents. This evolution has changed the role of the browser into an operating system rather than a simple document renderer. Chromium is built like an operating system to run these applications in a safe and robust way, using multiple OS processes to isolate web sites from each other and from the browser itself. This improves robustness because each process runs in its own address space, is scheduled by the operating system, and can fail independently. Users can also view the resource usage of each process in Chromium's Task Manager.

There are many ways that a web browser could be segmented into different OS processes, and choosing the best architecture depends on many factors, including stability, resource usage, and observations from actual usage. Chromium supports four different process models to allow experimentation, with a default model that best fits most users.

Supported models

Chromium supports four different models that affect how the browser allocates pages into renderer processes. By default, Chromium uses a separate OS process for each instance of a web site the user visits. However, users can specify command-line switches when starting Chromium to select one of the other architectures: one process for all instances of a web site, one process for each group of connected tabs, or everything in a single process. These models differ in whether they reflect the origin of the content, the relationships between tabs, or both. This section discusses each model in greater detail; caveats in Chromium's current implementation are described later in this document.

Process-per-site-instance

By default, Chromium creates a renderer process for each instance of a site the user visits. This ensures that pages from different sites are rendered independently, and that separate visits to the same site are also isolated from each other. Thus, failures (e.g., renderer crashes) or heavy resource usage in one instance of a site will not affect the rest of the browser. This model is based on both the origin of the content and relationships between tabs that might script each other. As a result, two tabs may display pages that are rendered in the same process, while navigating to a cross-site page in a given tab may switch the tab's rendering process. (Note that there are important caveats in Chromium's current implementation, discussed in the Caveats section below.)

Concretely, we define a "site" as a registered domain name (e.g., google.com or bbc.co.uk) plus a scheme (e.g., https://). This is similar to the origin defined by the Same Origin Policy, but it groups subdomains (e.g., mail.google.com and docs.google.com) and ports (e.g., http://foo.com:8080) into the same site. This is necessary to allow pages that are in different subdomains or ports of a site to access each other via Javascript, which is permitted by the Same Origin Policy if they set their document.domain variables to be identical.

A "site instance" is a collection of connected pages from the same site. We consider two pages as connected if they can obtain references to each other in script code (e.g., if one page opened the other in a new window using Javascript).

Strengths

- Isolates content from different sites. This provides a meaningful form of fate sharing for web content, where pages are isolated from failures caused by other web sites.
- Isolates independent tabs showing the same site. Visiting the same site independently in different tabs will create different processes. This will prevent contention and failures in one instance from affecting other instances.

Weaknesses

- More memory overhead. In most workloads, this model will create more renderer processes than the process-per-site model described below. While this increases stability and may add opportunities for parallelism, it also increases memory overhead.
- More complex to implement. Unlike process-per-tab and single-process, this model requires complex logic to support swapping processes in a tab when it navigates between sites, as well as proxying a small set of JavaScript actions that are permitted between origins, such as postMessage. (For more on this issue and our ongoing efforts to support it fully, see the Caveats section below and our Site Isolation project page.)

Process-per-site

Chromium also supports a process model that isolates different sites from each other, but groups all instances of the same site into the same process. To use this model, users should specify a --process-per-site command-line switch when starting Chromium. This creates fewer renderer processes, trading some robustness for lower memory overhead. This model is based on the origin of the content and not the relationships between tabs.

Strengths

- Isolates content from different sites. As in the process-per-site-instance model, pages from different sites will not share fate.
- Less memory overhead. This model is likely to create fewer concurrent processes than the process-per-site-instance and process-per-tab models. This may be desirable to reduce Chromium's memory footprint.

Weaknesses

- Can result in large renderer processes. Sites like google.com host a wide variety of applications that may be open concurrently in the browser, all of which would be rendered in the same process. Thus, resource contention and failures in these applications could affect many tabs, making the browser seem less responsive. It is unfortunately hard to identify site boundaries at a finer granularity than the registered domain name without breaking backwards compatibility.
- More complex to implement. Like the process-per-site-instance model, this requires logic for swapping processes during navigation and proxying some JavaScript interactions.

Process-per-tab

The process-per-site-instance and process-per-site models both consider the origin of the content when creating renderer processes. Chromium also supports a simpler model which dedicates one renderer process to each group of script-connected tabs. This model can be selected using the --process-per-tab command-line switch.

Specifically, we refer to a set of tabs with script connections to each other as a browsing instance, which also corresponds to a "unit of related browsing contexts" from the HTML5 spec. This set consists of a tab and any other tabs that it opens using Javascript code. Such tabs must be rendered in the same process to allow Javascript calls to be made between them (most commonly between pages from the same origin).

Strengths

- Simple to understand. Each tab has one renderer process dedicated to it that does not change over time.

Weaknesses

- Leads to undesirable fate sharing between pages. If the user navigates a tab in a browsing instance to a different web site, the new page will share fate with any other pages in the browsing instance.

It is worth noting that Chromium still forces process swaps within a tab in some situations in the process-per-tab model, when it is required for security. For example, normal web pages are not allowed to share a process with privileged pages like Settings and the New Tab Page. As a result, this model is not significantly simpler in practice than process-per-site-instance.

Single process

Finally, for the purposes of comparison, Chromium supports a single process model that can be enabled using the --single-process command-line switch. In this model, both the browser and rendering engine are run within a single OS process.

The single process model provides a baseline for measuring any overhead that the multi-process architectures impose. It is not a safe or robust architecture, as any renderer crash will cause the loss of the entire browser process. It is designed for testing and development purposes, and it may contain bugs that are not present in the other architectures.

Sandboxes and plug-ins

In each of the multi-process architectures, Chromium's renderer processes are executed within a sandboxed process that has limited access to the user's computer. These processes do not have direct access to the user's filesystem, display, or most other resources. Instead, they gain access to permitted resources only through the browser process, which can impose security policies on this access. As a result, Chromium's browser process can mitigate the damage that an exploited rendering engine can do.

Browser plug-ins, such as Flash and Silverlight, are also executed in their own processes, and some plug-ins like Flash even run within Chromium's sandbox. In each of the multi-process architectures that Chromium supports, there is one process instance for each type of active plug-in. Thus, all Flash instances run in the same process, regardless of which sites or tabs they appear in.

Caveats

This section lists a few caveats with Chromium's current implementation of the process models, along with their implications.

- Most renderer-initiated navigations within a tab do not yet lead to process swaps. If the user follows a link, submits a form, or is redirected by a script, Chromium will not attempt to switch renderer processes in the tab if the navigation is cross-site. Chromium only swaps renderer processes for browser-initiated cross-site navigations, such as typing a URL in the location bar or following a bookmark. As a result, pages from different sites may be rendered in the same process, even in the process-per-site-instance and process-per-site models. This is likely to change in future versions of Chromium as part of the Site Isolation project.

However, there is a mechanism web pages can use to suggest that a link points to an unrelated page and can be safely rendered in a different process. If a link has the rel=noreferrer target=_blank attributes, then Chromium will typically render it in a different process.

- Subframes are currently rendered in the same process as their parent page. Although cross-site subframes do not have script access to their parents and could safely be rendered in a separate process, Chromium does not yet render them in their own processes. Similar to the first caveat, this means that pages from different sites may be rendered in the same process. This will likely change in future versions of Chromium.
- There is a limit to the number of renderer processes that Chromium will create. This prevents the browser from overwhelming the user's computer with too many processes. The limit is is proportional to the amount of memory on the computer, and may be as high as 80 processes. Because of the limit, a single renderer process may be dedicated to multiple sites. This reuse is currently done at random, but future versions of Chromium may apply heuristics to more intelligently allocate sites to renderer processes.

Implementation notes

Two classes in Chromium represent the abstractions needed for the various process models: BrowsingInstance and SiteInstance.

The BrowsingInstance class represents a set of script-connected tabs within the browser, also known as a unit of related browsing contexts in the HTML 5 spec. In the process-per-tab model, we create a renderer process for each BrowsingInstance.

The SiteInstance class represents a set of connected pages from the same site. It is a subdivision of pages within a BrowsingInstance, and it is important that there is only one SiteInstance per site within a BrowsingInstance. In the process-per-site-instance model, we create a renderer process for each SiteInstance. To implement process-per-site, we ensure that all SiteInstances from the same site end up in the same process.

Academic Papers

Isolating Web Programs in Modern Browser Architectures

Charles Reis, Steven D. Gribble (both authors at UW + Google)

Eurosys 2009. Nuremberg, Germany, April 2009.

Abstract:

Many of today's web sites contain substantial amounts of client-side code, and consequently, they act more like programs than simple documents. This creates robustness and performance challenges for web browsers. To give users a robust and responsive platform, the browser must identify program boundaries and provide isolation between them.

We provide three contributions in this paper. First, we present abstractions of web programs and program instances, and we show that these abstractions clarify how browser components interact and how appropriate program boundaries can be identified. Second, we identify backwards compatibility tradeoffs that constrain how web content can be divided into programs without disrupting existing web sites. Third, we present a multi-process browser architecture that isolates these web program instances from each other, improving fault tolerance, resource management, and performance. We discuss how this architecture is implemented in Google Chrome, and we provide a quantitative performance evaluation examining its benefits and costs.

Security Architecture of the Chromium Browser Adam Barth, Collin Jackson, Charles Reis, and The Google Chrome Team

Stanford Technical Report, September 2008.

Abstract:

Most current web browsers employ a monolithic architecture that combines "the user" and "the web" into a single protection domain. An attacker who exploits an arbitrary code execution vulnerability in such a browser can steal sensitive files or install malware. In this paper, we present the security architecture of Chromium, the open-source browser upon which Google Chrome is built. Chromium has two modules in separate protection domains: a browser kernel, which interacts with the operating system, and a rendering engine, which runs with restricted privileges in a sandbox. This architecture helps mitigate high-severity attacks without sacrificing compatibility with existing web sites. We define a threat model for browser exploits and evaluate how the architecture would have mitigated past vulnerabilities.

Profile Architecture

This page details an ongoing design refactoring, started in January 2012.

Note: As of Jun 2013, this doc needs updating. The classes have been renamed (s/ProfileKeyed/BrowserContextKeyed/) and moved into components/browser_context_keyed_service.

Chromium has lots of features that hook into a **Profile**, a bundle of data about the current user and the current chrome session that can span multiple browser windows. When Chromium first started, the profile had only a few moving parts: the cookie jar, the history database, the bookmark database, and things to do with user preferences. In the three years of the Chromium Project, Profile became the join point for every feature, leading to things like Profile::GetInstantPromoCounter() or Profile::GetHostContentSettingsMap(). As of this writing there are 58 pure virtual methods that start with "Get" in Profile.

Profile should be a minimal reference, a sort of handle object that doesn't own the world.

Design Goals

- **We must be able to move to the new architecture piece-wise.** One service and feature at a time. We can not stop the world and convert everything in one operation. As of this writing, we've moved 19 services out of Profile.
  - We should only make small modifications at the callsite where a Profile is used to get the service in question.
- We must fix Profile shutdown. When we started and only had a few objects hanging off of Profile, manual ordering was acceptable for destruction. Now we have over seventy five components and we know that our manual destruction ordering is incorrect as written today. We can not rely on manual ordering when we have so many components.
- We must allow features to be compiled in and out. Now that we have chromium variants that don't contain all the features in a standard Windows/Mac/Linux Google Chrome build, we need a way to allow these variants to compile without #ifdefing profile.h and profile_impl.h into a mess. These variants also have their own services that they'd like to provide. (Letting chromium variants add their own services also touches on why we can't rely on manual ordering in Profile shutdown.)
  - Stretch goal: Separate features go in their own .a/.so files to further minimize our ridiculous linking time.

BrowserContextKeyedServiceFactory

The Old Way: Profile interface and ProfileImpl

In the previous design, services were fetched through an accessor on Profile:

```
class ProfileImpl {
  public:
    virtual FooService* GetFooService();
  private:
    scoped_ptr<FooService> foo_service_;
};
```

In the previous system, Profile was an interface with mostly pure virtual accessors. There were separate versions of Profile for Normal, Incognito and Testing profiles.

In this world, the Profile was the center of all activity. The profile owned all of its service and handed them out. Profile destruction was according to whatever order the services were listed in ProfileImpl. There wasn't a way for another variant to add its own services (or leave out ones it didn't need) without modifying the Profile interface.

The New Way: BrowserContextKeyedServiceFactory

Instead of having the Profile own FooService, we have a dedicated singleton FooServiceFactory, like this minimal one:

```
class FooServiceFactory : public BrowserContextKeyedServiceFactory {
 public:
  static FooService* GetForProfile(Profile* profile);

  static FooServiceFactory* GetInstance();

 private:
  friend struct DefaultSingletonTraits<FooServiceFactory>;

  FooServiceFactory();
  virtual ~FooServiceFactory();

  // BrowserContextKeyedServiceFactory:
  virtual BrowserContextKeyedService* BuildServiceInstanceFor(
  content::BrowserContext* context) const OVERRIDE;
};
```

We have a generalized BrowserContextKeyedServiceFactory which performs most of the work of associating a profile with an object provided by your BuildServiceInstanceFor() method. The BrowserContextKeyedServiceFactory provides an interface for you to override while managing the lifetime of your Service object in response to Profile lifetime events and making sure your service is shut down before services it depends on.

An absolutely minimal factory will supply the following methods:

- A static GetInstance() method that refers to your Factory as a Singleton.
- A constructor that associates this BrowserContextKeyedServiceFactory with the ProfileDependencyManager singleton, and makes DependsOn() declarations.
- A GetForProfile() method that wraps BrowserContextKeyedServiceFactory, casting the result back to whatever type you need to return.
- A BuildServiceInstanceFor() method which is called once by the framework for each |profile|, which must return a proper instance of your service.

In addition, BrowserContextKeyedServiceFactory provides these other knobs for how you can control behavior:

- RegisterUserPrefs() is called once per Profile during initialization and is where you can place any user pref registration.
- By default, BCKSF return NULL when given an Incognito profile.
  - If you override ServiceRedirectedInIncognito() to return true, it will return the associated normal Profile's service.
  - If you override ServiceHasOwnInstanceInIncognito() to return true, it will create a new service for the incognito profile.
- By default, BCKSF will lazily create your service. If you override ServiceIsCreatedWithProfile() to return true, your service will be created alongside the profile.
- BCKSF gives you multiple ways to control behavior during unit tests. See the header for more details.
- BCKSF gives you a way to augment and tweak the shutdown and deallocation behavior.

A Few Types of Factories

Not all objects have the same lifecycle and memory management. The previous paragraph was a major simplification; there is a base class BrowserContextKeyedBaseFactory that defines the most general dependency stuff while BrowserContextKeyedServiceFactory is a specialization that deals with normal objects. There is a second RefcountedBrowserContextKeyedServiceFactory that gives slightly different semantics and storage for RefCountedThreadSafe objects.

A Brief Interlude About Complexity

So the above, from an implementation standpoint is significantly more complex than what came before it. Is all this really worth it?

Yes.

We absolutely have to address the interdependency of services. As it stands today, we do not shut down profiles after they are no longer needed in multiprofile mode because our crash rate when shutting down a profile is too high to ship to users. We have about 75 components that plug into the profile lifecycle and their dependency graph is complex enough that our naive manual ordering can not handle the complexity. All of the overrideable behavior above exists because it was implemented per service, ad hoc and copy pasted.

We likewise need to make it easy for other chromium variants to add their own features/compile features out of their build.

Dependency Management Overview

With that in mind, let's look at how dependency management works. There is a single ProfileDependencyManager singleton, which is what is alerted to Profile creation and destruction. A PKSF will register and unregister itself with the ProfileDependencyManager. The job of the ProfileDependencyManager is to make sure that individual services are created and destroyed in a safe ordering.

Consider the case of these three service factories:

```
AlphaServiceFactory::AlphaServiceFactory()
    : BrowserContextKeyedServiceFactory(ProfileDependencyManager::GetInstance()) {
}

BetaServiceFactory::BetaServiceFactory()
    : BrowserContextKeyedServiceFactory(ProfileDependencyManager::GetInstance()) {
  DependsOn(AlphaServiceFactory::GetInstance());
     }

GammaServiceFactory::GammaServiceFactory()
    : BrowserContextKeyedServiceFactory(ProfileDependencyManager::GetInstance()) {
  DependsOn(BetaServiceFactory::GetInstance());
     }
```

The explicitly stated dependencies in this simplified graph mean that the only valid creation order for services is [Alpha, Beta, Gamma] and the destruction order is [Gamma, Beta, Alpha]. The above is all you, as a user of the framework, have to do to specify dependencies.

Behind the scenes, ProfileDependencyManager takes the stated dependency edges, performs a Kahn topological sort, and uses that in CreateProfileServices() and DestroyProfileServices().

The Five Minute Tutorial of How to Convert Your Code

1. ***Make Your Existing FooService derive from BrowserContextKeyedService.**
2. **If possible, make your FooService no longer refcounted.** Most of the refcounted objects that hang off of Profile appear to be that way because they aren't using base::bind/WeakPtrFactory instead of needing to own data on multiple threads. (In case you have a real reason for being a RefCountedThreadSafe, such as being accessed on multiple threads, derive your factory from RefcountedBrowserContextKeyedServiceFactory and everything should just work.)
3. **Build a simple FooServiceFactory derived from BrowserContextKeyedServiceFactory.** Your FooServiceFactory will be the main access point consumers will ask for FooService. BrowserContextKeyedServiceFactory gives you a bunch of virtual methods that control behavior.
   i. BrowserContextKeyedService *BrowserContextKeyedServiceFactory::BuildServiceInstanceFor(content::BrowserContext* context) is the only required method. Given a BrowserContext handle, return a valid FooService.
   ii. You can control the incognito behavior with ServiceRedirectedInIncognito() and ServiceHasOwnInstanceInIncognito().
4. **Add your service to the** EnsureBrowserContextKeyedServiceFactoriesBuilt() **list in** chrome_browser_main_extra_parts_profiles.cc.

5. **Understand shutdown behavior.** For historical reasons, we have a two phase deletion process:
    i. Every BrowserContextKeyedService will first have its Shutdown() method called. Use this method to drop weak references to the Profile or other service objects.
    ii. Every BrowserContextKeyedService is deleted and its destructor is run. Minimal work should be done here. Attempts to call any *ServiceFactory::GetForProfile() will cause an assertion in debug mode.
6. Change each instance of "profile->*GetFooService()" to "FooServiceFactory::GetForProfile(profile*)". If you need an example of what the above looks like, try looking at these patches:
7. r100516: A simple example, adding a new ProfileKeyedService. This shows off a minimal ServiceFactory subclass.
8. r104806: plugin_prefs_factory.h gives an example of how to deal with things that are (and have to stay) refcounted. This patch also shows off how to move your preferences into your ProfileKeyedServiceFactory.

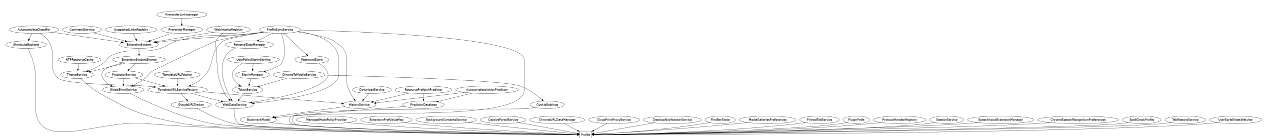
Debugging Tips

Using the dependency visualizer

Chrome has a built in method to dump the profile dependency graph to a file in GraphViz format. When you run chrome with the command line flag --dump-browser-context-graph, chrome will write the dependency information to your /path/to/profile/browser-context-dependencies.dot file. You can then convert this text file with dot, which is part of GraphViz:

```
dot -Tpng /path/to/profile/browser-context-dependencies.dot > png-file.png
```

This will give you a visual graph like this (generated January 23rd, 2012, click through for full size):



Graph as of Aug 15, 2012

Crashes at Shutdown

If you get a stack that looks like this:

```
ProfileDependencyManager::AssertProfileWasntDestroyed()
ProfileKeyedServiceFactory::GetServiceForProfile()
MyServiceFactory::GetForProfile()
... [Probably a bunch of frames] ...
OtherService::~OtherService()
ProfileKeyedServiceFactory::ProfileDestroyed()
ProfileDependencyManager::DestroyProfileServices()
ProfileImpl::~ProfileImpl()
```

The problem is that OtherService is improperly depending on MyService. The framework asserts if you try to use a Shutdown()ed component.

Safe Browsing

Browsing Protection

When Safe Browsing is enabled, all URLs will be checked before the content is allowed to begin loading. URLs are checked against two lists: malware and phishing. Depending on which list is matched we show a different warning message on the interstitial page.

Checking the safe browsing database is a multistep process. The URL is hashed and a synchronous check against the in-memory prefix list is done. If no match is found, the URL is considered safe immediately. If the prefix matches, an asynchronous request is made to the safe browsing servers for a list of all full hashes matching that prefix. Once the list is returned, the full hash is compared against the list and the URL request can be continued or cancelled. For more information, you may check the full description of the Safe Browsing Protocol.

Resource Handlers

Whenever a resource is requested, the ResourceDispatcherHost will create a chain of ResourceHandlers. For each event in the loading of the resource, each handler can choose to cancel the request, defer the request (to do some asynchronous work before deciding what to do), or continue (letting the next handler in the chain have a chance to decide). The SafeBrowsingResourceHandler is created at the head of the chain so that it has first say over whether to allow loading a resource. If safe browsing is disabled, the SafeBrowsingResourceHandler is simply not added to the chain, and thus no browsing-related safe browsing actions occur.

Safe Browsing Interstitial Page

When a resource is marked as unsafe the resource request is paused and an interstitial page (SafeBrowsingBlockingPage) is displayed. The user can choose to continue anyway, which will resume the resource request, or to go back, which will cancel the resource request and return to the previous page.

Threat Details Collection

If the interstitial is for a hit in the threat list (including malware, phishing, and UwS), the page is http (not https), and the tab is not in an incognito window, there is an opt-in option to send extra details about the the unsafe resources for further analysis.

When the interstitial appears an IPC is sent to the renderer process to collect details from the DOM. The data consists of a tree of the URLs for the various frames, iframes, scripts and embeds.

If the checkbox is checked when the user chooses dismisses the interstitial page, various extra details will be collected asynchronously on the browser side. First the History service is queried to get the list of redirects involved in all the URLs, then the Cache is queried to get the headers for each of the requests for those URLs, and finally the report will be sent.

Download Protection

URL Checking

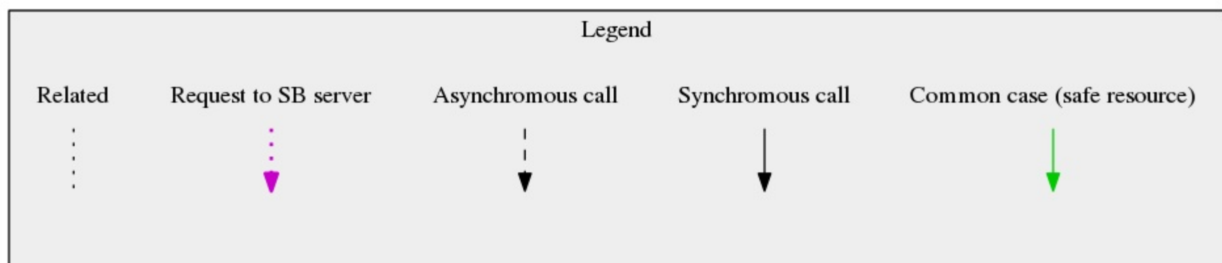
The download checks operate in a similar manner to the browsing ones, though with some changes due to the different nature of downloads. It is not known that a resource request will be a download until the headers are received, therefore all downloads also go through the browsing checks. For the same reason, we cannot check the redirect URLs as we go along like is done in the browsing tests. Instead the chain of redirects is saved in the URLRequest object and once we begin the download checks, all the URLs in the chain will be checked simultaneously. Since downloads are less latency sensitive

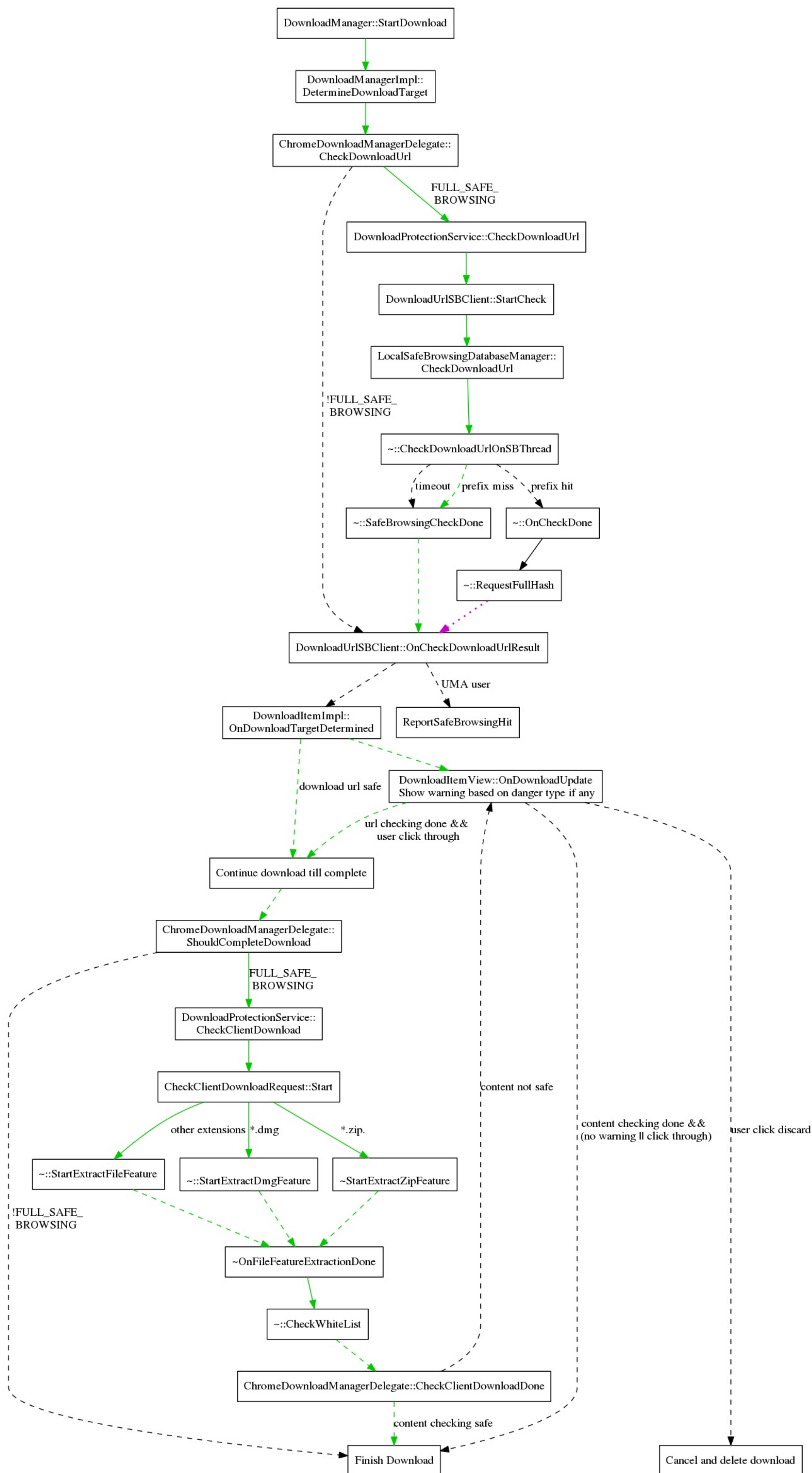
than page loads, we also dispense with the in-memory database and the caching of full hash results. Finally, the check is done in parallel to the download rather than pausing the download request until the checks are done, however the file will be given a temporary name until the checks complete. If a download is flagged as malicious, the item in the download bar will be replaced with a warning and buttons to keep or discard the file. If discard is chosen, the request will be cancelled and the file deleted. If the file is kept, it will be renamed to its actual name (with .crdownload if the download is still in progress).

Hash Checking

As the file downloads, we also compute a hash of the file data. Once the file has completed downloading this hash is checked against the download digest list. Currently we are evaluating the usefulness of the hash check so no UI is displayed.

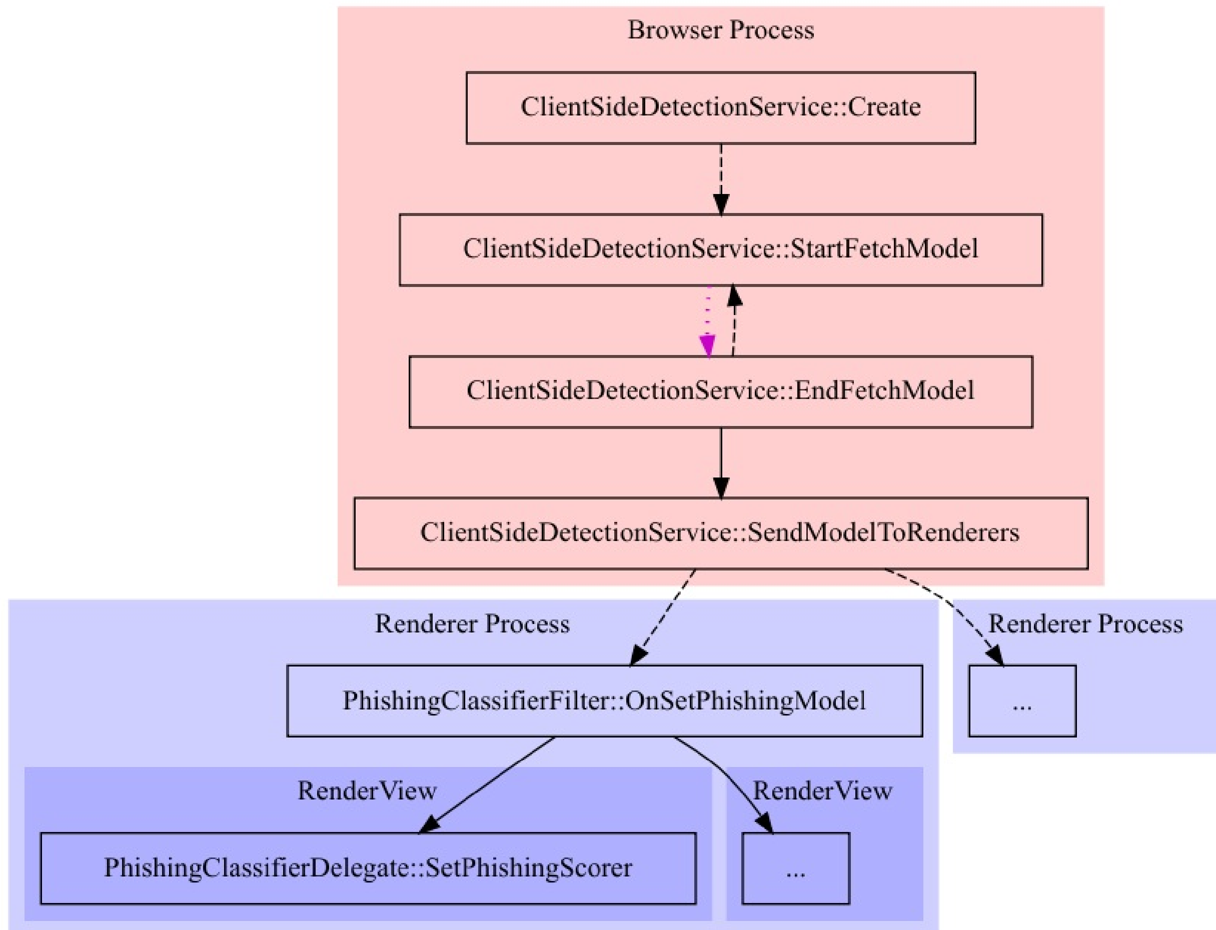
This is an overview of the code flow of handling a download. Some details are omitted to keep the size reasonable. This is an overview of the code flow of handling a request. Some details are omitted to keep the size reasonable. The green line indicates the common case where loading a non-malware page only requires a synchronous check to the in-memory safe browsing database. The dashed lines indicate asynchronous calls. The dotted magenta lines indicates a request to Google's Safe Browsing server.




DownloadManager::StartDownload
DownloadManagerImpl::
DetermineDownloadTarget
ChromeDownloadManagerDelegate::
CheckDownloadUrl
FULL_SAFE_
BROWSING
DownloadProtectionService::CheckDownloadUrl
DownloadUrlSBClient::StartCheck
LocalSafeBrowsingDatabaseManager::
CheckDownloadUrl
!FULL_SAFE_
BROWSING
~::CheckDownloadUrlOnSBThread
timeout
prefix miss
prefix hit
~::SafeBrowsingCheckDone
~::OnCheckDone
~::RequestFullHash
DownloadUrlSBClient::OnCheckDownloadUrlResult
UMA user
DownloadItemImpl::
OnDownloadTargetDetermined
ReportSafeBrowsingHit
DownloadItemView::OnDownloadUpdate
Show warning based on danger type if any
download url safe
url checking done &&
user click through
Continue download till complete
ChromeDownloadManagerDelegate::
ShouldCompleteDownload
FULL_SAFE_
BROWSING
DownloadProtectionService::
CheckClientDownload
CheckClientDownloadRequest::Start
content not safe
content checking done &&
(no warning || click through)
user click discard
other extensions
*.dmg
*.zip.
~::StartExtractFileFeature
~::StartExtractDmgFeature
~StartExtractZipFeature
!FULL_SAFE_
BROWSING
~OnFileFeatureExtractionDone
~::CheckWhiteList
ChromeDownloadManagerDelegate::CheckClientDownloadDone
content checking safe
Finish Download
Cancel and delete download

Client Side Phishing Detection

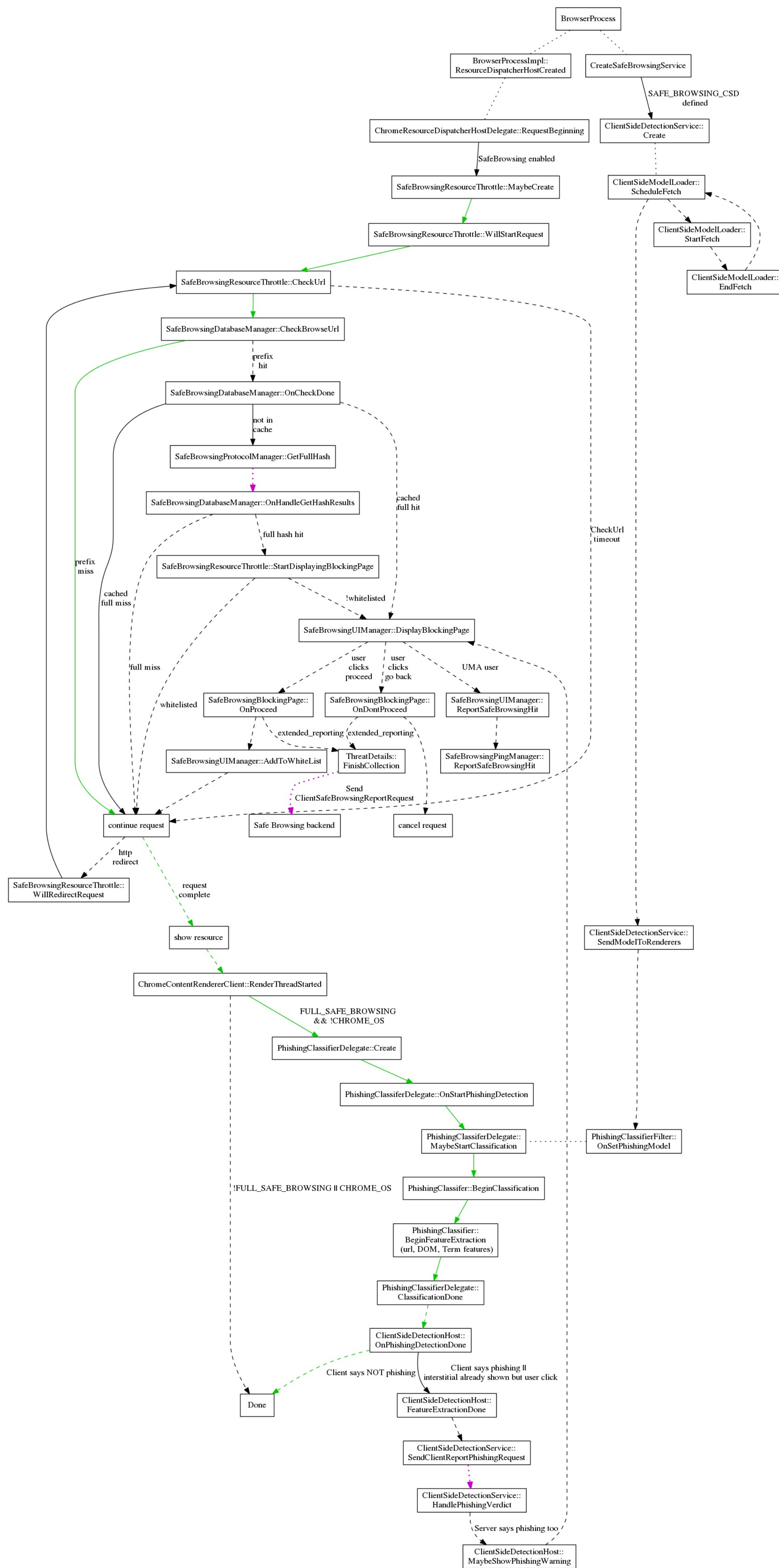
Client Side Phishing Detection runs a detection model on pages the user visits to try to detect phishing pages that are not in the safe browsing lists. On startup, and periodically afterwards, the ClientSideDetectionService will fetch an updated model. The model is sent in an IPC to every Render Process, then assigned to PhishingClassifierDelegate associated with each RenderView. This allows the classification to be done in the render process, which has access to the page text.



Resource Request Flow

This is an overview of the code flow of handling a request. Some details are omitted to keep the size reasonable. The green line indicates the common case where loading a non-malware page only requires a synchronous check to the in-memory safe browsing database. The dashed lines indicate asynchronous calls. The dotted magenta lines indicates a request to Google's Safe Browsing server.

Safe Browsing Resource Request Diagram


BrowserProcess
BrowserProcessImpl::
ResourceDispatcherHostCreated
CreateSafeBrowsingService
SAFE_BROWSING_CSD
defined
ChromeResourceDispatcherHostDelegate::RequestBeginning
ClientSideDetectionService::
Create
SafeBrowsing enabled
SafeBrowsingResourceThrottle::MaybeCreate
ClientSideModelLoader::
ScheduleFetch
SafeBrowsingResourceThrottle::WillStartRequest
ClientSideModelLoader::
StartFetch
ClientSideModelLoader::
EndFetch
SafeBrowsingResourceThrottle::CheckUrl
SafeBrowsingDatabaseManager::CheckBrowseUrl
prefix
hit
SafeBrowsingDatabaseManager::OnCheckDone
not in
cache
SafeBrowsingProtocolManager::GetFullHash
SafeBrowsingDatabaseManager::OnHandleGetHashResults
cached
full hit
CheckUrl
timeout
full hash hit
prefix
miss
SafeBrowsingResourceThrottle::StartDisplayingBlockingPage
cached
full miss
!whitelisted
SafeBrowsingUIManager::DisplayBlockingPage
user
clicks
proceed
user
clicks
go back
UMA user
full miss
whitelisted
SafeBrowsingBlockingPage::
OnProceed
SafeBrowsingBlockingPage::
OnDontProceed
SafeBrowsingUIManager::
ReportSafeBrowsingHit
extended_reporting
extended_reporting
SafeBrowsingUIManager::AddToWhiteList
ThreatDetails::
FinishCollection
SafeBrowsingPingManager::
ReportSafeBrowsingHit
Send
ClientSafeBrowsingReportRequest
continue request
Safe Browsing backend
cancel request
http
redirect
request
complete
SafeBrowsingResourceThrottle::
WillRedirectRequest
show resource
ClientSideDetectionService::
SendModelToRenderers
ChromeContentRendererClient::RenderThreadStarted
FULL_SAFE_BROWSING
&& !CHROME_OS
PhishingClassifierDelegate::Create
PhishingClassifierDelegate::OnStartPhishingDetection
PhishingClassifierDelegate::
MaybeStartClassification
PhishingClassifierFilter::
OnSetPhishingModel
!FULL_SAFE_BROWSING || CHROME_OS
PhishingClassifer::BeginClassification
PhishingClassifier::
BeginFeatureExtraction
(url, DOM, Term features)
PhishingClassifierDelegate::
ClassificationDone
ClientSideDetectionHost::
OnPhishingDetectionDone
Client says NOT phishing
Client says phishing ||
interstitial already shown but user click
Done
ClientSideDetectionHost::
FeatureExtractionDone
ClientSideDetectionService::
SendClientReportPhishingRequest
ClientSideDetectionService::
HandlePhishingVerdict
Server says phishing too
ClientSideDetectionHost::
MaybeShowPhishingWarning

Metrics

Safe browsing histograms use the "SB2." prefix. Histograms for older versions used "SB.". There are also a few safe browsing UserMetrics (filter on "SB"), and safe browsing Rappor metrics (starts with "interstitial").

Safe Browsing Database

The SafeBrowsingService is responsible for updating the various databases used by safe browsing. TODO(mattm): provide more details about database format and update process.

Sandbox

Security is one of the most important goals for Chromium. The key to security is understanding: we can only truly secure a system if we fully understand its behaviors with respect to the combination of all possible inputs in all possible states. For a codebase as large and diverse as Chromium, reasoning about the combined behavior of all its parts is nearly impossible. The sandbox objective is to provide hard guarantees about what ultimately a piece of code can or cannot do no matter what its inputs are.

Sandbox leverages the OS-provided security to allow code execution that cannot make persistent changes to the computer or access information that is confidential. The architecture and exact assurances that the sandbox provides are dependent on the operating system. This document covers the Windows implementation as well as the general design. The Linux implementation is described here, the OSX implementation here.

If you don't feel like reading this whole document you can read the Sandbox FAQ instead. A description of what the sandbox does and doesn't protect against may also be found in the FAQ.

Design principles

- **Do not re-invent the wheel**: It is tempting to extend the OS kernel with a better security model. Don't. Let the operating system apply its security to the objects it controls. On the other hand, it is OK to create application-level objects (abstractions) that have a custom security model.
- **Principle of least privilege**: This should be applied both to the sandboxed code and to the code that controls the sandbox. In other words, the sandbox should work even if the user cannot elevate to super-user.
- **Assume sandboxed code is malicious code**: For threat-modeling purposes, we consider the sandbox compromised (that is, running malicious code) once the execution path reaches past a few early calls in the main() function. In practice, it could happen as soon as the first external input is accepted, or right before the main loop is entered.
- **Be nimble**: Non-malicious code does not try to access resources it cannot obtain. In this case the sandbox should impose near-zero performance impact. It's ok to have performance penalties for exceptional cases when a sensitive resource needs to be touched once in a controlled manner. This is usually the case if the OS security is used properly.
- **Emulation is not security**: Emulation and virtual machine solutions do not by themselves provide security. The sandbox should not rely on code emulation, code translation, or patching to provide security.

Sandbox windows architecture

The Windows sandbox is a user-mode only sandbox. There are no special kernel mode drivers, and the user does not need to be an administrator in order for the sandbox to operate correctly. The sandbox is designed for both 32-bit and 64-bit processes and has been tested on all Windows OS flavors between Windows 7 and Windows 10, both 32-bit and 64-bit.

Sandbox operates at process-level granularity. Anything that needs to be sandboxed needs to live on a separate process. The minimal sandbox configuration has two processes: one that is a privileged controller known as the broker, and one or more sandboxed processes known as the target. Throughout the documentation and the code these two terms are used with that precise connotation. The sandbox is provided as a static library that must be linked to both the broker and the target executables.

The broker process

In Chromium, the broker is always the browser process. The broker, is in broad terms, a privileged controller/supervisor of the activities of the sandboxed processes. The responsibilities of the broker process are:

1. Specify the policy for each target process
2. Spawn the target processes
3. Host the sandbox policy engine service

4. Host the sandbox interception manager
5. Host the sandbox IPC service (to the target processes)
6. Perform the policy-allowed actions on behalf of the target process

The broker should always outlive all the target processes that it spawned. The sandbox IPC is a low-level mechanism (different from Chromium's IPC) that is used to transparently forward certain windows API calls from the target to the broker: these calls are evaluated against the policy. The policy-allowed calls are then executed by the broker and the results returned to the target process via the same IPC. The job of the interceptions manager is to patch the windows API calls that should be forwarded via IPC to the broker.

The target process

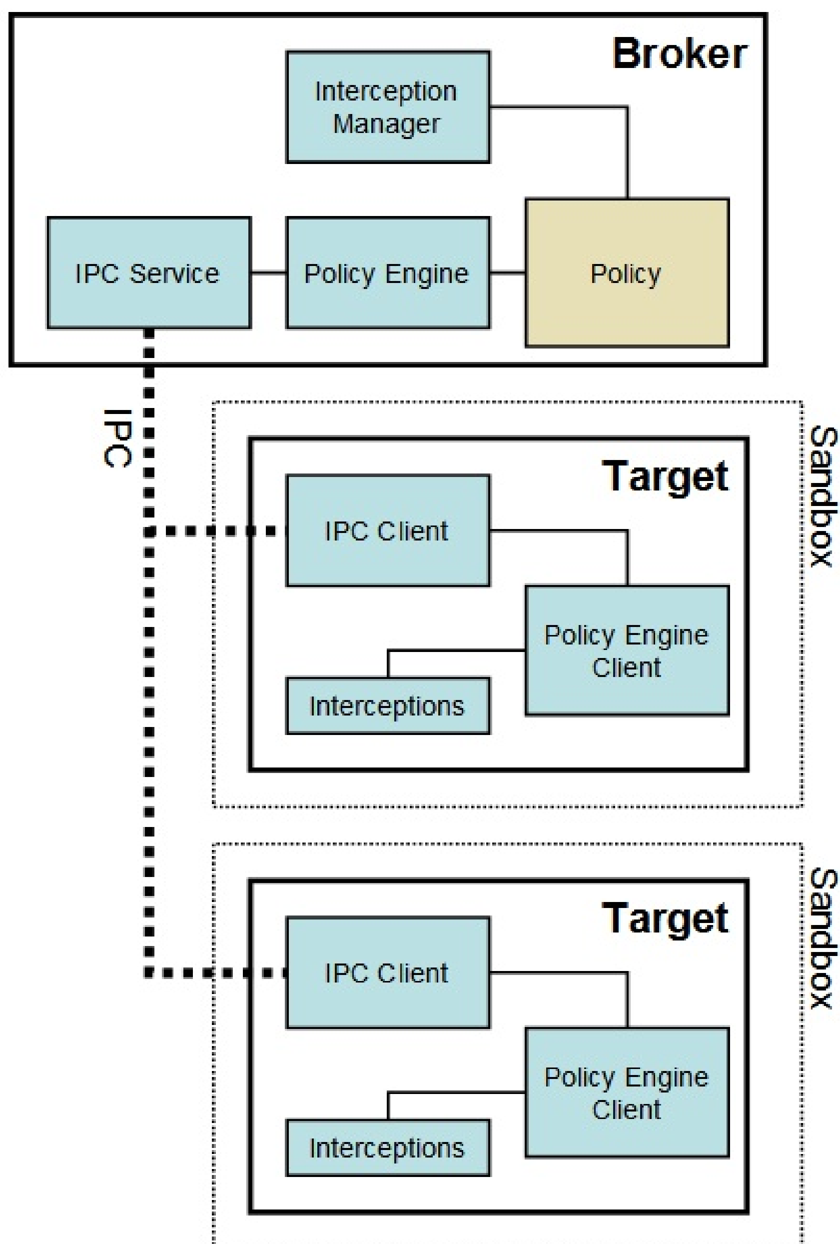
In Chromium, the renderers are always target processes, unless the --no-sandbox command line has been specified for the browser process. The target process hosts all the code that is going to run inside the sandbox, plus the sandbox infrastructure client side:

1. All code to be sandboxed
2. The sandbox IPC client
3. The sandbox policy engine client
4. The sandbox interceptions

Items 2,3 and 4 are part of the sandbox library that is linked with the code to be sandboxed.

The interceptions (also known as hooks) are how Windows API calls are forwarded via the sandbox IPC to the broker. It is up to the broker to re-issue the API calls and return the results or simply fail the calls. The interception + IPC mechanism does not provide security; it is designed to provide compatibility when code inside the sandbox cannot be modified to cope with sandbox restrictions. To save unnecessary IPCs, policy is also evaluated in the target process before making an IPC call, although this is not used as a security guarantee but merely a speed optimization.

It is the expectation that in the future most plugins will run inside a target process.



Sandbox restrictions

At its core, the sandbox relies on the protection provided by four Windows mechanisms:

- A restricted token
- The Windows job object
- The Windows desktop object
- Windows Vista and above: The integrity levels

These mechanisms are highly effective at protecting the OS, its configuration, and the user's data provided that:

- All the securable resources have a better than null security descriptor. In other words, there are no critical resources with misconfigured security.
- The computer is not already compromised by malware. *Third party software does not weaken the security of the system.

Note that extra mitigations above and beyond this base/core will be described in the "Process Mitigations" section below.

The token

One issue that other similar sandbox projects face is how restricted can the token and job be while still having a properly functioning process. For the Chromium sandbox, the most restrictive token for Windows XP takes the following form:

Regular Groups

Logon SID : mandatory

All other SIDs : deny only, mandatory

Restricted Groups

S-1-0-0 : mandatory

Privileges

None

With the caveats described above, it is near impossible to find an existing resource that the OS will grant access with such a token. As long as the disk root directories have non-null security, even files with null security cannot be accessed. In Vista, the most restrictive token is the same but it also includes the low integrity level label. The Chromium renderer normally runs with this token, which means that almost all resources that the renderer process uses have been acquired by the Browser and their handles duplicated into the renderer process.

Note that the token is not derived from anonymous or from the guest token; it is derived from the user's token and thus associated to the user logon. As a result, any auditing that the system or the domain has in place can still be used.

By design, the sandbox token cannot protect the following non-securable resources:

- Mounted FAT or FAT32 volumes: The security descriptor on them is effectively null. Malware running in the target can read and write to these volumes as long it can guess or deduce their paths.
- TCP/IP: The security of TCP/IP sockets in Windows 2000 and Windows XP (but not in Vista) is effectively null. It might be possible for malicious code in the target to send and receive network packets to any host.
- More information about the Windows token object can be found at [02].

The Job object

The target process also runs under a Job object. Using this Windows mechanism, some interesting global restrictions that do not have a traditional object or security descriptor associated with them are enforced:

- Forbid per-use system-wide changes using SystemParametersInfo(), which can be used to swap the mouse buttons or set the screen saver timeout
- Forbid the creation or switch of Desktops
- Forbid changes to the per-user display configuration such as resolution and primary display
- No read or write to the clipboard
- Forbid Windows message broadcasts
- Forbid setting global Windows hooks (using SetWindowsHookEx())
- Forbid access to the global atoms table
- Forbid access to USER handles created outside the Job object
- One active process limit (disallows creating child processes)

Chromium renderers normally run with all these restrictions active. Each renderers run in its own Job object. Using the Job object, the sandbox can (but currently does not) prevent:

- Excessive use of CPU cycles
- Excessive use of memory

- Excessive use of IO

More information about Windows Job Objects can be found in [1].

The alternate desktop

The token and the job object define a security boundary: that is, all processes with the same token and in the same job object are effectively in the same security context. However, one not-well-understood fact is that applications that have windows on the same desktop are also effectively in the same security context because the sending and receiving of window messages is not subject to any security checks. Sending messages across desktops is not allowed. This is the source of the infamous "shatter" attacks, which is why services should not host windows on the interactive desktop. A Windows desktop is a regular kernel object that can be created and assigned a security descriptor.

In a standard Windows installation, at least two desktops are attached to the interactive window station; the regular (default) desktop, and the logon desktop. The sandbox creates a third desktop that is associated to all target processes. This desktop is never visible or interactive and effectively isolates the sandboxed processes from snooping the user's interaction and from sending messages to windows operating at more privileged contexts.

The only disadvantage of an alternate desktop is that it uses approximately 4MB of RAM from a separate pool, possibly more on Vista.

More information about Window Stations

The integrity levels

Integrity levels are available on Windows Vista and later versions. They don't define a security boundary in the strict sense, but they do provide a form of mandatory access control (MAC) and act as the basis of Microsoft's Internet Explorer sandbox.

Integrity levels are implemented as a special set of SID and ACL entries representing five levels of increasing privilege: untrusted, low, medium, high, system. Access to an object may be restricted if the object is at a higher integrity level than the requesting token. Integrity levels also implement User Interface Privilege Isolation, which applies the rules of integrity levels to window messages exchanged between different processes on the same desktop.

By default, a token can read an object of a higher integrity level, but not write to it. Most desktop applications run at medium integrity (MI), while less trusted processes like Internet Explorer's protected mode and our own sandbox run at low integrity (LI). A low integrity mode token can access only the following shared resources:

- Read access to most files
- Write access to %USER PROFILE%\AppData\LocalLow
- Read access to most of the registry
- Write access to HKEY_CURRENT_USER\Software\AppDataLow Clipboard (copy and paste for certain formats)
- Remote procedure call (RPC)
- TCP/IP Sockets
- Window messages exposed via ChangeWindowMessageFilter
- Shared memory exposed via LI (low integrity) labels
- COM interfaces with LI (low integrity) launch activation rights
- Named pipes exposed via LI (low integrity) labels

You'll notice that the previously described attributes of the token, job object, and alternate desktop are more restrictive, and would in fact block access to everything allowed in the above list. So, the integrity level is a bit redundant with the other measures, but it can be seen as an additional degree of defense-in-depth, and its use has no visible impact on performance or resource usage.

More information on integrity levels can be found at [03].

Process mitigation policies

Most process mitigation policies can be applied to the target process by means of SetProcessMitigationPolicy. The sandbox uses this API to set various policies on the target process for enforcing security characteristics.

Relocate Images:

- >= Win8
- Address-load randomization (ASLR) on all images in process (and must be supported by all images).

Heap Terminate:

- >= Win8
- Terminates the process on Windows heap corruption.

Bottom-up ASLR:

- >= Win8
- Sets random lower bound as minimum user address for the process.

High-entropy ASLR:

- >= Win8
- Increases randomness range for bottom-up ASLR to 1TB.

Strict Handle Checks:

- >= Win8
- Immediately raises an exception on a bad handle reference.

Win32k.sys lockdown:

- >= Win8
- ProcessSystemCallDisablePolicy, which allows selective disabling of system calls available from the target process.
- Renderer processes now have this set to DisallowWin32kSystemCalls which means that calls from user mode that are serviced by win32k.sys are no longer permitted. This significantly reduces the kernel attack surface available from a renderer. See here for more details.

App Container (low box token):

- >= Win8
- In Windows this is implemented at the kernel level by a Low Box token which is a stripped version of a normal token with limited privilege (normally just SeChangeNotifyPrivilege and SeIncreaseWorkingSetPrivilege), running at Low integrity level and an array of "Capabilities" which can be mapped to allow/deny what the process is allowed to do (see MSDN for a high level description). The capability most interesting from a sandbox perspective is denying is access to the network, as it turns out network checks are enforced if the token is a Low Box token and the INTERNET_CLIENT Capability is not present.
- The sandbox therefore takes the existing restricted token and adds the Low Box attributes, without granting any Capabilities, so as to gain the additional protection of no network access from the sandboxed process.

Disable Font Loading:

- >= Win10
- ProcessFontDisablePolicy

Disable Image Load from Remote Devices:

- >= Win10 TH2
- ProcessImageLoadPolicy
- E.g. UNC path to network resource.

Disable Image Load of "mandatory low" (low integrity level):

- >= Win10 TH2
- ProcessImageLoadPolicy

- E.g. temporary internet files.

Extra Disable Child Process Creation:

- >= Win10 TH2
- If the Job level <= JOB_LIMITED_USER, set PROC_THREAD_ATTRIBUTE_CHILD_PROCESS_POLICY to PROCESS_CREATION_CHILD_PROCESS_RESTRICTED via UpdateProcThreadAttribute().
- This is an extra layer of defense, given that Job levels can be broken out of. [REF: ticket, Project Zero blog.]

Other caveats

The operating system might have bugs. Of interest are bugs in the Windows API that allow the bypass of the regular security checks. If such a bug exists, malware will be able to bypass the sandbox restrictions and broker policy and possibly compromise the computer. Under Windows, there is no practical way to prevent code in the sandbox from calling a system service.

In addition, third party software, particularly anti-malware solutions, can create new attack vectors. The most troublesome are applications that inject dlls in order to enable some (usually unwanted) capability. These dlls will also get injected in the sandbox process. In the best case they will malfunction, and in the worst case can create backdoors to other processes or to the file system itself, enabling specially crafted malware to escape the sandbox.

Sandbox policy

The actual restrictions applied to a target process are configured by a policy. The policy is just a programmatic interface that the broker calls to define the restrictions and allowances. Four functions control the restrictions, roughly corresponding to the four Windows mechanisms:

- TargetPolicy::SetTokenLevel()
- TargetPolicy::SetJobLevel()
- TargetPolicy::SetIntegrityLevel()
- TargetPolicy::SetDesktop()

The first three calls take an integer level parameter that goes from very strict to very loose; for example, the token level has 7 levels and the job level has 5 levels. Chromium renderers are typically run with the most strict level in all four mechanisms. Finally, the last (desktop) policy is binary and can only be used to indicate if a target is run on an alternate desktop or not.

The restrictions are by design coarse in that they affect all securable resources that the target can touch, but sometimes a more finely-grained resolution is needed. The policy interface allows the broker to specify exceptions. An exception is a way to take a specific Windows API call issued in the target and proxy it over to the broker. The broker can inspect the parameters and re-issue the call as is, re-issue the call with different parameters, or simply deny the call. To specify exceptions there is a single call: AddRule. The following kinds of rules for different Windows subsystems are supported at this time:

- Files
- Named pipes
- Process creation
- Registry
- Synchronization objects

The exact form of the rules for each subsystem varies, but in general rules are triggered based on a string pattern. For example, a possible file rule is:

```
AddRule(SUBSYS_FILES, FILES_ALLOW_READONLY, L"c:\\temp\\app_log\\d*.dmp")
```

This rule specifies that access will be granted if a target wants to open a file, for read-only access as long as the file matches the pattern expression; for example c:\temp\app_log\domino.dmp is a file that satisfies the pattern. Consult the header files for an up-to-date list of supported objects and supported actions.

Rules can only be added before each target process is spawned, and cannot be modified while a target is running, but different targets can have different rules.

Target bootstrapping

Targets do not start executing with the restrictions specified by policy. They start executing with a token that is very close to the token the regular user processes have. The reason is that during process bootstrapping the OS loader accesses a lot of resources, most of them are actually undocumented and can change at any time. Also, most applications use the standard CRT provided with the standard development tools; after the process is bootstrapped the CRT needs to initialize as well and there again the internals of the CRT initialization are undocumented.

Therefore, during the bootstrapping phase the process actually uses two tokens: the lockdown token which is the process token as is and the initial token which is set as the impersonation token of the initial thread. In fact the actual SetTokenLevel definition is:

```
SetTokenLevel(TokenLevel initial, TokenLevel lockdown)
```

After all the critical initialization is done, execution continues at main() or WinMain(), here the two tokens are still active, but only the initial thread can use the more powerful initial token. It is the target's responsibility to discard the initial token when ready. This is done with a single call:

```
LowerToken()
```

After this call is issued by the target the only token available is the lockdown token and the full sandbox restrictions go into effect. The effects of this call cannot be undone. Note that the initial token is a impersonation token only valid for the main thread, other threads created in the target process use only the lockdown token and therefore should not attempt to obtain any system resources subject to a security check.

The fact that the target starts with a privileged token simplifies the explicit policy since anything privileged that needs to be done once, at process startup can be done before the LowerToken() call and does not require to have rules in the policy.

> **Important**
>
> Make sure any sensitive OS handles obtained with the initial token are closed before calling LowerToken(). Any leaked handle can be abused by malware to escape the sandbox.

References

[01] Richter, Jeffrey "Make Your Windows 2000 Processes Play Nice Together With Job Kernel Objects"

http://www.microsoft.com/msj/0399/jobkernelobj/jobkernelobj.aspx

[02] Brown, Keith "What Is a Token" (wiki)

http://alt.pluralsight.com/wiki/default.aspx/Keith.GuideBook/WhatIsAToken.htm

[03] Windows Integrity Mechanism Design (MSDN)

http://msdn.microsoft.com/en-us/library/bb625963.aspx

Sandbox FAQ

What is the sandbox?

The sandbox is a C++ library that allows the creation of sandboxed processes — processes that execute within a very restrictive environment. The only resources sandboxed processes can freely use are CPU cycles and memory. For example, sandboxes processes cannot write to disk or display their own windows. What exactly they can do is controlled by an explicit policy. Chromium renderers are sandboxed processes.

What does and doesn't it protect against?

The sandbox limits the severity of bugs in code running inside the sandbox. Such bugs cannot install persistent malware in the user's account (because writing to the filesystem is banned). Such bugs also cannot read and steal arbitrary files from the user's machine.

(In Chromium, the renderer processes are sandboxed and have this protection. Plugins for Chromium do not yet run inside the sandbox, because many are designed with the assumption that they have full access to the local system. Also note that Chromium renderer processes are isolated from the system, but not yet from the web. Therefore, domain-based data isolation is not yet provided).

The sandbox cannot provide any protection against bugs in system components such as the kernel it is running on.

Is the sandbox like what you get with the Java VM?

Yeah, kind of... except that to take advantage of the Java sandbox, you must rewrite your code to use Java. With our sandbox you can add sandboxing to your existing C/C++ applications. Because the code is not executed inside a virtual machine, you get native speed and direct access to the Windows API.

Do I need to install a driver or kernel module? Does the user need to be Administrator?

No and no. The sandbox is a pure user-mode library, and any user can run sandboxed processes.

How can you do this for C++ code if there is no virtual machine?

We leverage the Windows security model. In Windows, code cannot perform any form of I/O (be it disk, keyboard, or screen) without making a system call. In most system calls, Windows performs some sort of security check. The sandbox sets things up so that these security checks fail for the kinds of actions that you don't want the sandboxed process to perform. In Chromium, the sandbox is such that all access checks should fail.

So how can a sandboxed process such as a renderer accomplish anything?

Certain communication channels are explicitly open for the sandboxed processes; the processes can write and read from these channels. A more privileged process can use these channels to do certain actions on behalf of the sandboxed process. In Chromium, the privileged process is usually the browser process.

Doesn't Vista have similar functionality?

Yes. It's called integrity levels (ILs). The sandbox detects Vista and uses integrity levels, as well. The main difference is that the sandbox also works well under Windows XP. The only application that we are aware of that uses ILs is Internet Explorer 7. In other words, leveraging the new Vista security features is one of the things that the sandbox library does for you.

This is very neat. Can I use the sandbox in my own programs?

Yes. The sandbox does not have any hard dependencies on the Chromium browser and was designed to be used with other Internet-facing applications. The main hurdle is that you have to split your application into at least two interacting processes. One of the processes is privileged and does I/O and interacts with the user; the other is not privileged at all and does untrusted data processing.

Isn't that a lot of work?

Possibly. But it's worth the trouble, especially if your application processes arbitrary untrusted data. Any buffer overflow or format decoding flaw that your code might have won't automatically result in malicious code compromising the whole computer. The sandbox is not a security silver bullet, but it is a strong last defense against nasty exploits.

Should I be aware of any gotchas?

Well, the main thing to keep in mind is that you should only sandbox code that you fully control or that you fully understand. Sandboxing third-party code can be very difficult. For example, you might not be aware of third-party code's need to create temporary files or display warning dialogs; these operations will not succeed unless you explicitly allow them. Furthermore, third-party components could get updated on the end-user machine with new behaviors that you did not anticipate.

How about COM, Winsock, or DirectX — can I use them?

For the most part, no. We recommend against using them before lock-down. Once a sandboxed process is locked down, use of Winsock, COM, or DirectX will either malfunction or outright fail.

What do you mean by before lock-down? Is the sandboxed process not locked down from the start?

No, the sandboxed process does not start fully secured. The sandbox takes effect once the process makes a call to the sandbox method LowerToken(). This allows for a period during process startup when the sandboxed process can freely get hold of critical resources, load libraries, or read configuration files. The process should call LowerToken() as soon as feasible and certainly before it starts interacting with untrusted data.

Note: Any OS handle that is left open after calling LowerToken() can be abused by malware if your process gets infected. That's why we discourage calling COM or other heavyweight APIs; they leave some open handles around for efficiency in later calls.

So what APIs can you call?

There is no master list of safe APIs. In general, structure your code such that the sandboxed code reads and writes from pipes or shared memory and just does operations over this data. In the case of Chromium, the entire WebKit code runs this way, and the output is mostly a rendered bitmap of the web pages. You can use Chromium as inspiration for your own memory- or pipe-based IPC.

But can't malware just infect the process at the other end of the pipe or shared memory?

Yes, it might, if there's a bug there. The key point is that it's easier to write and analyze a correct IPC mechanism than, say, a web browser engine. Strive to make the IPC code as simple as possible, and have it reviewed by others.

OSX Sandboxing Design

This document describes the process sandboxing mechanism used on Mac OS X.

Background

Sandboxing treats a process as a hostile environment which at any time can be compromised by a malicious attacker via buffer overruns or other such attack vectors. Once compromised, the goal is to allow the process in question access to as few resources of the user's machine as possible, above and beyond the standard file-system access control and user/group process controls enforced by the kernel.

See the overview document for goals and general architectural diagrams.

Implementation

On Mac OS X versions starting from Leopard, individual processes can have their privileges restricted using the sandbox(7) facility of BSD, also referred to in some Apple documentation as "Seatbelt". This is made up of a single API call, sandbox_init(), which sets the access restrictions of a process from that point on. This means that previously opened file descriptors continue working even if the new privileges would deny access to newly created descriptors. We can use this to our advantage by setting up everything correctly at the start of the process then cutting off all access before we expose the renderer to any 3rd party input (html, etc).

Seatbelt does not place restrictions on memory allocation, threading, or access to previously opened OS facilities. As a result, this shouldn't impose any additional requirements or drastically alter our IPC designs.

OS X provides additional protection against buffer overflows. In Leopard, the stack is marked as non-executable memory and thus less susceptible as an attack vector for executing malicious code. This doesn't prevent against buffer overruns in the heap, but for 64-bit apps, Leopard disallows any attempts to execute code unless that portion of memory is explicitly marked as executable. As we move towards 64-bit render processes in the future, this will be another attractive security feature.

sandbox_init() has supports for both predefined sandbox access restrictions and sandbox profile scripts which provide finer grained control.

Chromium uses custom sandbox profiles defined in .sb files in the source tree.

The following profiles are defined (paths relative to root of source directory):

- content/common/common.sb - used for common setup for all sandboxes.
- content/renderer/renderer.sb - used for the extension & renderer processes. Enables access to the font server.
- chrome/browser/utility.sb - used by the utility process. Allows access to a single configurable directory.
- content/browser/worker.sb - used by the worker process. Most restrictive - no file system access apart from loading system libraries.
- chrome/browser/nacl_loader.sb - used for running Native Client untrusted (i.e., "user") code.

One sticky point we run into is that the sandboxed process calls through to OS X system APIs. There is no documentation available about which privileges each API needs, such as whether they need access to on-disk files, or call other APIs to which the sandbox restricts access. Our approach to date has been to "warm up" any problematic API calls before turning the sandbox on. This means that we call through to the API, to allow it to cache whatever resource it needs. For example, color profiles and shared libraries can be loaded from disk before we "lock down" the process.

SandboxInitWrapper::InitializeSandbox() is the main entry point for initializing the Sandbox, it performs the following steps:

- Determines if the current process type needs to be sandboxed and if so, which sandbox configuration to use.

- "Warm up" relevant system APIs by calling through to sandbox::SandboxWarmup() .
- Enable the Sandbox by calling through to sandbox::EnableSandbox() .

Diagnostics

The OS X sandbox implementation supports the following flags:

- --no-sandbox - Disable the sandbox entirely.
- --enable-sandbox-logging - Verbose information about which system calls are blocked is logged to syslog.

Debugging Chrome on OS X contains more documentation on debugging and diagnostic tools provided by the Mac OS X sandbox API.

Additional Reading

- http://reverse.put.as/wp-content/uploads/2011/09/Apple-Sandbox-Guide-v1.0.pdf
- http://www.318.com/techjournal/?p=107
- sandbox man page (man 7 sandbox)
- System sandbox files can be found under one of the following paths (depending on the OS Version):
  - /Library/Sandbox/Profiles
  - /System/Library/Sandbox/Profiles
  - /usr/share/sandbox

The Security Architecture of the Chromium Browser

Most current web browsers employ a monolithic architecture that combines "the user" and "the web" into a single protection domain. An attacker who exploits an arbitrary code execution vulnerability in such a browser can steal sensitive files or install malware. In this paper, we present the security architecture of Chromium, the open-source browser upon which Google Chrome is built. Chromium has two modules in separate protection domains: a browser kernel, which interacts with the operating system, and a rendering engine, which runs with restricted privileges in a sandbox. This architecture helps mitigate high-severity attacks without sacrificing compatibility with existing web sites. We define a threat model for browser exploits and evaluate how the architecture would have mitigated past vulnerabilities.

The Security Architecture of the Chromium Browser

Adam Barth, Collin Jackson, Charles Reis, and The Google Chrome Team Technical Report 2008

More Stanford web security research

Startup

Chrome is (mostly) shipped as a single executable that knows how to run as all the interesting sorts of processes we use. Here's an overview of how that works.

1. First there's the platform-specific entry point: wWinMain() on Windows, main() on Linux. This lives in chrome/app/chrome*exe_main**. On Mac and Windows, that function loads modules as described later, while on Linux it does very little, and all of them call into:
2. ChromeMain(), which is the place where cross-platform code that needs to run in all Chrome processes lives. It lives in chrome/app/chrome_main*. For example, here is where we call initializers for modules like logging and ICU. We then examine the internal --process-type switch and dispatch to:
3. A process-type-specific main function such as BrowserMain() (for the outer browser process) or RendererMain() (for a tab-specific renderer process).

Platform-specific entry points

Windows

On Windows we build the bulk of Chrome as a DLL. (XXX: why?) wWinMain loads chrome.dll, does some other random stuff (XXX: why?) and calls ChromeMain in the DLL.

Mac

Mac is also packaged as a framework and an executable, but they're linked together: main() calls ChromeMain() directly. There is also a second entry point, in chrome_main_app_mode_mac.mm, for app mode shortcuts: "On Mac, one can't make shortcuts with command-line arguments. Instead, we produce small app bundles which locate the Chromium framework and load it, passing the appropriate data." This executable also calls ChromeMain().

Linux

On Linux due to the sandbox we launch subprocesses by repeatedly forking from a helper process. This means that new subprocesses don't enter through main() again, but instead resume from clones in the middle of startup. The initial launch of the helper process still executes the normal startup path, so any initialization that happens in ChromeMain() will have been run for all subprocesses but they will all share the same initialization.

Threading

Overview

Chromium is a very multithreaded product. We try to keep the UI as responsive as possible, and this means not blocking the UI thread with any blocking I/O or other expensive operations. Our approach is to use message passing as the way of communicating between threads. We discourage locking and threadsafe objects. Instead, objects live on only one thread, we pass messages between threads for communication, and we use callback interfaces (implemented by message passing) for most cross-thread requests.

The Thread object is defined in base/threading/thread.h. In general you should probably use one of the existing threads described below rather than make new ones. We already have a lot of threads that are difficult to keep track of. Each thread has a MessageLoop (see base/message_loop/message_loop.h) that processes messages for that thread. You can get the message loop for a thread using the Thread.message_loop() function. More details about MessageLoop can be found in Anatomy of Chromium MessageLoop.

Existing threads

Most threads are managed by the BrowserProcess object, which acts as the service manager for the main "browser" process. By default, everything happens on the UI thread. We have pushed certain classes of processing into these other threads. It has getters for the following threads:

- **ui_thread**: Main thread where the application starts up.
- **io_thread**: This thread is somewhat mis-named. It is the dispatcher thread that handles communication between the browser process and all the sub-processes. It is also where all resource requests (web page loads) are dispatched from (see Multi-process Architecture).
- **file_thread**: A general process thread for file operations. When you want to do blocking filesystem operations (for example, requesting an icon for a file type, or writing downloaded files to disk), dispatch to this thread.
- **db_thread**: A thread for database operations. For example, the cookie service does sqlite operations on this thread. Note that the history database doesn't use this thread yet.
- **safe_browsing_thread**

Several components have their own threads:

- **History**: The history service object has its own thread. This might be merged with the db_thread above. However, we need to be sure that things happen in the correct order -- for example, that cookies are loaded before history since cookies are needed for the first load, and history initialization is long and will block it.
- **Proxy service**: See net/http/http_proxy_service.cc.
- **Automation proxy**: This thread is used to communicate with the UI test program driving the app.

Keeping the browser responsive

As hinted in the overview, we avoid doing any blocking I/O on the UI thread to keep the UI responsive. Less apparent is that we also need to avoid blocking I/O on the IO thread. The reason is that if we block it for an expensive operation, say disk access, then IPC messages don't get processed. The effect is that the user can't interact with a page. Note that asynchronous/overlapped IO are fine.

Another thing to watch out for is to not block threads on one another. Locks should only be used to swap in a shared data structure that can be accessed on multiple threads. If one thread updates it based on expensive computation or through disk access, then that slow work should be done without holding on to the lock. Only when the result is available should the

lock be used to swap in the new data. An example of this is in PluginList::LoadPlugins (src/content/common/plugin_list.cc). If you must use locks, here are some best practices and pitfalls to avoid.

In order to write non-blocking code, many APIs in Chromium are asynchronous. Usually this means that they either need to be executed on a particular thread and will return results via a custom delegate interface, or they take a base::Callback<> object that is called when the requested operation is completed. Executing work on a specific thread is covered in the PostTask section below.

Getting stuff to other threads

base::Callback<>, Async APIs, and Currying

A base::Callback<> (see the docs in callback.h) is templated class with a Run() method. It is a generalization of a function pointer and is created by a call to base::Bind. Async APIs often will take a base::Callback<> as a means to asynchronously return the results of an operation. Here is an example of a hypothetical FileRead API.

```
void ReadToString(const std::string& filename, const base::Callback<void(const std::string&)>& on_read);
```

```
void DisplayString(const std::string& result) {
  LOG(INFO) << result;
}
```

```
void SomeFunc(const std::string& file) {
  ReadToString(file, base::Bind(&DisplayString));
};
```

In the example above, base::Bind takes the function pointer &DisplayString and turns it into a base::Callback. The type of the generated base::Callback<> is inferred from the arguments. Why not just pass the function pointer directly? The reason is base::Bind allows the caller to adapt function interfaces and/or attach extra context via Currying (http://en.wikipedia.org/wiki/Currying). For instance, if we had a utility function DisplayStringWithPrefix that took an extra argument with the prefix, we use base::Bind to adapt the interface as follows.

```
void DisplayStringWithPrefix(const std::string& prefix, const std::string& result) {
  LOG(INFO) << prefix << result;
}
```

```
void AnotherFunc(const std::string& file) {
  ReadToString(file, base::Bind(&DisplayStringWithPrefix, "MyPrefix: "));
};
```

This can be used in lieu of creating an adapter functions a small classes that holds prefix as a member variable. Notice also that the "MyPrefix: " argument is actually a const char*, while DisplayStringWithPrefix actually wants a const std::string&. Like normal function dispatch, base::Bind, will coerce parameters types if possible. See "How arguments are handled by base::Bind()" below for more details about argument storage, copying, and special handling of references.

PostTask

The lowest level of dispatching to another thread is to use the MessageLoop.PostTask and MessageLoop.PostDelayedTask (see base/message_loop/message_loop.h). PostTask schedules a task to be run on a particular thread. A task is defined as a base::Closure, which is a typedef for a base::Callback. PostDelayedTask schedules a task to be run after a delay on a particular thread. A task is represented by the base::Closure typedef, which contains a Run() function, and is created by calling base::Bind(). To process a task, the message loop eventually calls base::Closure's Run function, and then drops the reference to the task object. Both PostTask and PostDelayedTask take a tracked_objects::Location parameter, which is

used for lightweight debugging purposes (counts and primitive profiling of pending and completed tasks can be monitored in a debug build via the url about:objects). Generally the macro value FROM_HERE is the appropriate value to use in this parameter.

Note that new tasks go on the message loop's queue, and any delay that is specified is subject to the operating system's timer resolutions. This means that under Windows, very small timeouts (under 10ms) will likely not be honored (and will be longer). Using a timeout of 0 in PostDelayedTask is equivalent to calling PostTask, and adds no delay beyond queuing delay. PostTask is also used to do something on the current thread "sometime after the current processing returns to the message loop." Such a continuation on the current thread can be used to assure that other time critical tasks are not starved on this thread.

The following is an example of a creating a task for a function and posting it to another thread (in this example, the file thread):

```
void WriteToFile(const std::string& filename, const std::string& data);
BrowserThread::PostTask(BrowserThread::FILE, FROM_HERE,
                        base::Bind(&WriteToFile, "foo.txt", "hello world!"));
```

You should always use BrowserThread to post tasks between threads. Never cache MessageLoop pointers as it can cause bugs such as the pointers being deleted while you're still holding on to them. More information can be found here.

base::Bind() and class methods.

The base::Bind() API also supports invoking class methods as well. The syntax is very similar to calling base::Bind() on a function, except the first argument should be the object the method belongs to. By default, the object that PostTask uses must be a thread-safe reference-counted object. Reference counting ensures that the object invoked on another thread will stay alive until the task completes.

```
class MyObject : public base::RefCountedThreadSafe<MyObject> {
 public:
  void DoSomething(const std::string16& name) {
    thread_->message_loop()->PostTask(
       FROM_HERE, base::Bind(&MyObject::DoSomethingOnAnotherThread, this, name));
  }

  void DoSomethingOnAnotherThread(const std::string16& name) {
    ...
  }
 private:
  // Always good form to make the destructor private so that only RefCountedThreadSafe can access it.
  // This avoids bugs with double deletes.
  friend class base::RefCountedThreadSafe<MyObject>;

  ~MyObject();
  Thread* thread_;
};
```

If you have external synchronization structures that can completely insure that an object will always be alive while the task is waiting to execute, you can wrap the object pointer with base::Unretained() when calling base::Bind() to disable the refcounting. This will also allow using base::Bind() on classes that are not refcounted. Be careful when doing this!

How arguments are handled by base::Bind().

The arguments given to base::Bind() are copied into an internal InvokerStorage structure object (defined in base/bind_internal.h). When the function is finally executed, it will see copies of the arguments. This is important if your target function or method takes a const reference; the reference will be to a copy of the argument. If you need a reference to the original argument, you can wrap the argument with base::ConstRef(). Use this carefully as it is likely dangerous if

target of the reference cannot be guaranteed to live past when the task is executed. In particular, it is almost never safe to use base::ConstRef() to a variable on the stack unless you can guarantee the stack frame will not be invalidated until the asynchronous task finishes.

Sometimes, you will want to pass reference-counted objects as parameters (be sure to use RefCountedThreadSafe and not plain RefCounted as the base class for these objects). To ensure that the object lives throughout the entire request, the Closure generated by base::Bind must keep a reference to it. This can be done by passing scoped_refptr as the parameter type, or by wrapping the raw pointer with make_scoped_refptr():

```
class SomeParamObject : public base::RefCountedThreadSafe<SomeParamObject> {
 ...
};

class MyObject : public base::RefCountedThreadSafe<MyObject> {
 public:
  void DoSomething() {
    scoped_refptr<SomeParamObject> param(new SomeParamObject);
    thread_->message_loop()->PostTask(FROM_HERE
       base::Bind(&MyObject::DoSomethingOnAnotherThread, this, param));
  }
  void DoSomething2() {
    SomeParamObject* param = new SomeParamObject;
    thread_->message_loop()->PostTask(FROM_HERE
       base::Bind(&MyObject::DoSomethingOnAnotherThread, this,
                  make_scoped_refptr(param)));
  }
  // Note how this takes a raw pointer. The important part is that
  // base::Bind() was passed a scoped_refptr; using a scoped_refptr
  // here would result in an extra AddRef()/Release() pair.
  void DoSomethingOnAnotherThread(SomeParamObject* param) {
    ...
  }
};
```

If you want to pass the object without taking a reference on it, wrap the argument with base::Unretained(). Again, using this means there are external guarantees on the lifetime of the object, so tread carefully!

If your object has a non-trivial destructor that needs to run on a specific thread, you can use the following trait. This is needed since timing races could lead to a task completing execution before the code that posted it has unwound the stack.

```
class MyObject : public base::RefCountedThreadSafe<MyObject, BrowserThread::DeleteOnIOThread> {
```

Callback cancellation

There are 2 major reasons to cancel a task (in the form of a Callback):

- You want to do something later on your object, but at the time your callback runs, your object may have been destroyed.
- When input changes (e.g. user input), old tasks become unnecessary. For performance consideration, you should cancel them. See following about different approaches for cancellation.

Important notes about cancellation

It 's dangerous to cancel a task with owned parameters. See following example. (The example uses base::WeakPtr for cancellation, but the problem applies to all approaches).

```
class MyClass {
 public:
  // Owns |p|.
  void DoSomething(AnotherClass* p) {
    ...
  }
  WeakPtr<MyClass> AsWeakPtr() {
    return weak_factory_.GetWeakPtr();
  }
 private:
  base::WeakPtrFactory<MyClass> weak_factory_;
};
```

...

```
Closure cancelable_closure = Bind(&MyClass::DoSomething, object->AsWeakPtr(), p);
Callback<void(AnotherClass*)> cancelable_callback = Bind(&MyClass::DoSomething, object->AsWeakPtr());
```

...

```
void FunctionRunLater(const Closure& cancelable_closure,
                      const Callback<void(AnotherClass*)>& cancelable_callback) {
```

// Leak memory! cancelable_closure.Run(); cancelable_callback.Run(p); }

```
In FunctionRunLater, both Run() calls will leak p when object is already destructed. Using scoped_ptr can fix the bug
.
```c++
class MyClass {
 public:
  void DoSomething(scoped_ptr<AnotherClass> p) {
    ...
  }
  ...
};
```

base::WeakPtr and Cancellation [NOT THREAD SAFE]

You can use a base::WeakPtr and base::WeakPtrFactory (in base/memory/weak_ptr.h) to ensure that any invokes can not outlive the object they are being invoked on, without using reference counting. The base::Bind mechanism has special understanding for base::WeakPtr that will disable the task's execution if the base::WeakPtr has been invalidated. The base::WeakPtrFactory object can be used to generate base::WeakPtr instances that know about the factory object. When the factory is destroyed, all the base::WeakPtr will have their internal "invalidated" flag set, which will make any tasks bound to them to not dispatch. By putting the factory as a member of the object being dispatched to, you can get automatic cancellation.

NOTE: This only works when the task is posted to the same thread. Currently there is not a general solution that works for tasks posted to other threads. See the next section about CancelableTaskTracker for an alternative solution.
```

```
class MyObject {
 public:
  MyObject() : weak_factory_(this) {}

  void DoSomething() {
    const int kDelayMS = 100;
    MessageLoop::current()->PostDelayedTask(FROM_HERE,
        base::Bind(&MyObject::DoSomethingLater, weak_factory_.GetWeakPtr()),
        kDelayMS);
  }

  void DoSomethingLater() {
    ...
  }

 private:
  base::WeakPtrFactory<MyObject> weak_factory_;
};
```

CancelableTaskTracker

While base::WeakPtr is very helpful to cancel a task, it is not thread safe so can not be used to cancel tasks running on another thread. This is sometimes a performance critical requirement. E.g. We need to cancel database lookup task on DB thread when user changes inputed text. In this kind of situation CancelableTaskTracker is appropriate.

With CancelableTaskTracker you can cancel a single task with returned TaskId. This is another reason to use CancelableTaskTracker instead of base::WeakPtr, even in a single thread context.

CancelableTaskTracker has 2 Post methods doing the same thing as the ones in base::TaskRunner, with additional cancellation support.

```
class UserInputHandler : public base::RefCountedThreadSafe<UserInputHandler> {
  // Runs on UI thread.
  void OnUserInput(Input input) {
    CancelPreviousTask();
    DBResult* result = new DBResult();
    task_id_ = tracker_->PostTaskAndReply(
        BrowserThread::GetMessageLoopProxyForThread(BrowserThread::DB).get(),
        FROM_HERE,
        base::Bind(&LookupHistoryOnDBThread, this, input, result),
        base::Bind(&ShowHistoryOnUIThread, this, base::Owned(result)));
  }

  void CancelPreviousTask() {
    tracker_->TryCancel(task_id_);
  }

  ...

 private:
  CancelableTaskTracker tracker_;  // Cancels all pending tasks while destruction.
  CancelableTaskTracker::TaskId task_id_;
  ...
};
```

Since task runs on other threads, there's no guarantee it can be successfully canceled.

When TryCancel() is called:

- If neither task nor reply has started running, both will be canceled.
- If task is already running or has finished running, reply will be canceled.
- If reply is running or has finished running, cancelation is a noop.

Like base::WeakPtrFactory, CancelableTaskTracker will cancel all tasks on destruction.

Cancelable request (DEPRECATED)

Note. Cancelable request is deprecated. Please do not use it in new code. For canceling tasks running on the same thread, use WeakPtr. For canceling tasks running on a different thread, use CancelableTaskTracker.

A cancelable request makes it easier to make requests to another thread with that thread returning some data to you asynchronously. Like the revokable store system, it uses objects that track whether the originating object is alive. When the calling object is deleted, the request will be canceled to prevent invalid callbacks. Like the revokable store system, a user of a cancelable request has an object (here, called a "Consumer") that tracks whether it is alive and will auto-cancel any outstanding requests on deleting.

```
class MyClass {
  void MakeRequest() {
    frontend_service->StartRequest(some_input1, some_input2, this,
        // Use base::Unretained(this) if this may cause a refcount cycle.
        base::Bind(&MyClass:RequestComplete, this));
  }
  void RequestComplete(int status) {
    ...
  }

 private:
  CancelableRequestConsumer consumer_;
};
```

Note that the MyClass::RequestComplete, is bounded with base::Unretained(this) here.

The consumer also allows you to associate extra data with a request. Use CancelableRequestConsumer which will allow you to associate arbitrary data with the handle returned by the provider service when you invoke the request. The data will be automatically destroyed when the request is canceled.

A service handling requests inherits from CancelableRequestProvider. This object provides methods for canceling in-flight requests, and will work with the consumers to make sure everything is cleaned up properly on cancel. This frontend service just tracks the request and sends it to a backend service on another thread for actual processing. It would look like this:

```
class FrontendService : public CancelableRequestProvider {
  typedef base::Callback<void(int)> RequestCallbackType;

  Handle StartRequest(int some_input1, int some_input2,
   CallbackConsumer* consumer,
   const RequestCallbackType& callback) {
    scoped_refptr< CancelableRequest<FrontendService::RequestCallbackType> >
        request(new CancelableRequest(callback));
    AddRequest(request, consumer);

    // Send the parameters and the request to the backend thread.
    backend_thread_->PostTask(FROM_HERE,
        base::Bind(&BackendService::DoRequest, backend_, request,
                   some_input1, some_input2), 0);
    // The handle will have been set by AddRequest.
    return request->handle();
  }
};
```

The backend service runs on another thread. It does processing and forwards the result back to the original caller. It would look like this:

```
class BackendService : public base::RefCountedThreadSafe<BackendService> {
  void DoRequest(
    scoped_refptr< CancelableRequest<FrontendService::RequestCallbackType> >
        request,
    int some_input1, int some_input2) {
    if (request->canceled())
      return;

    ... do your processing ...

    // Execute ForwardResult() like you would do Run() on the base::Callback<>.
    request->ForwardResult(return_value);
  }
};
```

Also see the documentation for V8, which is the JavaScript engine used within Chromium.

Android

- JNI Binding
- Java Resource on Android
- WebView code organization

Java Resources on Android

Overview

Chrome for Android uses certain resources in Java code (e.g. Android layouts and associated strings or images). These resources are stored according to Android's resource directory structure within a Java root folder.

- content/public/android/java/res - Java resources available within content and anything that depends on content
- chrome/android/java/res - Java resources available within chrome and anything that depends on chrome
- ui/android/java/res - Java resources available within ui and anything that depends on ui

Java code can reference these resources in the normal way using a generated R class, being sure to qualify it with the correct package name.

```
// Use a resource from content
setImageResource(org.chromium.content.R.drawable.globe_favicon);

// Use a resource from chrome
setContentView(org.chromium.chrome.R.layout.month_picker);
```

How resources are packaged

While compiling the Java code in content, an R.java file is generated based on the Java resources in content. This R.java contains non-final constants and is used only while compiling content (and any non-APK target that depends on content) but is not included in the content jar.

When building an APK target, such as content_shell_apk, resources are merged from content, any other dependencies, and from content shell itself. These merged resources are processed and included in the APK. Based on these resources, a new R.java is generated with the correct resource -> ID mappings. This R.java is copied into the R packages needed by each dependency (e.g. org.chromium.content.R and org.chromium.content_shell.R), and all these copies are included in the APK.

This process closely follows Android's handling of resources in library projects, where content and chrome are the "libraries", though we don't use the SDK to compile our "libraries". Hence some of the same caveats apply. In particular, two resources with the same ID cannot coexist. The resource highest in the dependency chain (e.g. in content shell) will override the others (e.g. in content).

Supporting resources in gyp

To add resources to another Java root folder, add the variables has_java_resources, R_package, and R_package_relpath to the gyp target that builds that Java code. For example:

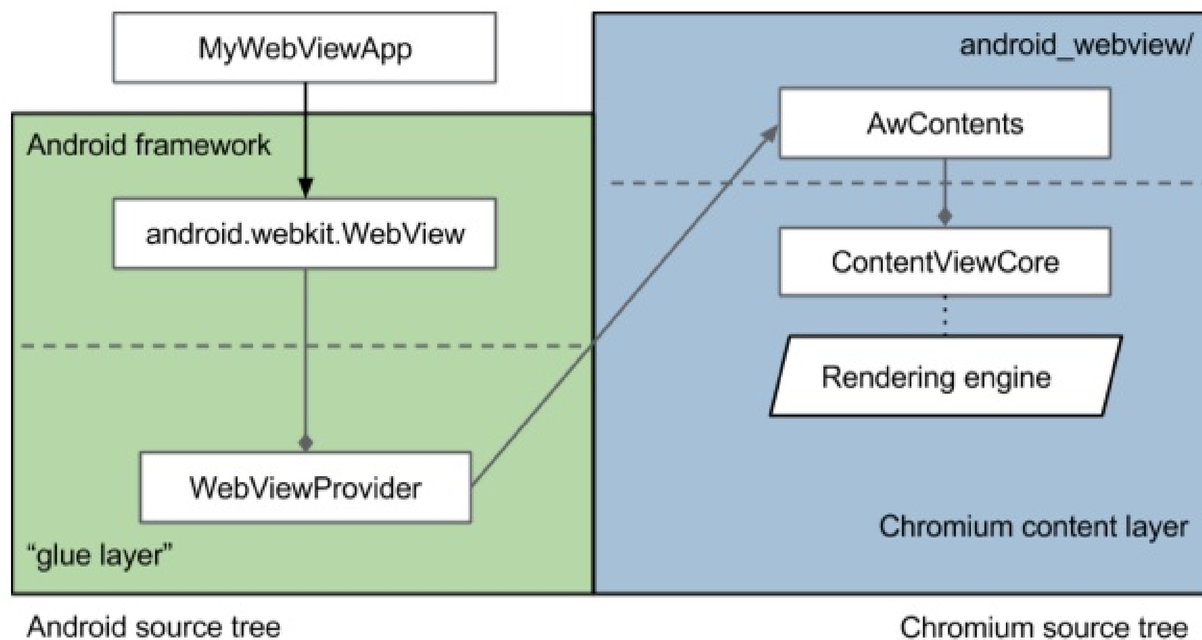
```
{
  'target_name': 'content_java',
  'type': 'none',
  'dependencies': [ ... ],
  'variables': {
    'package_name': 'content',
    'java_in_dir': '../content/public/android/java',

    # Support Java resources in content
    'has_java_resources': 1,
    'R_package': 'org.chromium.content',
    'R_package_relpath': 'org/chromium/content',
  },
  'includes': [ '../build/java.gypi' ],
},
```

Organization of code for Android WebView

android-webview-dev@chromium.org

Last updated Jul 2014.



A copy of the chromium source code (rooted from src.chromium.org/viewvc/chrome/trunk/src/) is mirrored into the Android build tree under external/chromium_org

CHROMIUM TREE ("upstream": trunk/src; "downstream": external/chromium_org)

- android_webview/java
  - Top-level entry-point into the chromium stack for webview.
  - Provides a semi-stable façade / wrapper over the chromium code stack for consumption by the downstream android tree. Unlike other chromium java code, the public interface to public classes in this package are considered first-class public API with consumers that are independently versioned outside of this git repository.
  - For the majority of WebView APIs the backend functionality is provided by the chromium content module, or ancillary browser components; for these features it forwards calls to the public Java APIs in other layers of the chromium stack, and to native counterparts over JNI via the android_webview/native folder.
- android_webview/native
  - Would be more appropriately named 'jni'. In general code in here is responsible for crossing the Java<->native boundary. Classes here tend to reflect their java counterpart naming, and primarily responsible for creational and object ownership and cleanup roles
  - Where possible avoid placing complex implementation logic in here instead factor out into more focused feature specific classes inside the android_webview/browser folder or as a component in the top-level components/ folder (e.g. if needs to be shared with Chrome browser).
  - By convention code in here should follow the same threading model as the public WebView API (that is, primarily executes on UI thread). Non-trivial interaction with BrowserThreads and IPC to renderer is extracted down to the browser/ folder.
- android_webview/browser
  - Provides non-trivial behavioral and functional classes for implementing features not directly exported by the content module public API.
  - By convention, any complex native threading code should be encapsulated here.
  - DEPS enforces that classes in this folder have no static dependency on the higher layer android_webview/native JNI wrappers. This aims to make the pieces in browser/ more modular and testable, and minimize the big ball of mud tendency.

- android_webview/renderer
  - Contains all code that logically resides in the renderer process. While WebView currently only supports single process mode, the browser/renderer separation is maintained as it is a useful architectural separation between the (implicitly trusted) java application and the (potentially untrusted) web-platform code.
- android_webview/lib
  - This is the main entry point to the libwebviewchromium.so native library. This is the top of the native code static dependency tree; no other module may depend on this.
- android_webview/common
  - Declares raw types etc that are shared by both android_webview/browser and android_webview/renderer
  - While single process mode makes it possible to share state via e.g. globals hidden in this module, resist this temptation. Instead common abstract types may be declared here, but the concrete instances should be constructed and passed in from the appropriate creational class in android_webview/lib.
  - As per chromium conventions, IPC message types are declared here.
- android_webview/public
  - Defines and exports abstract native interfaces for certain performance critical (e.g. rendering pipeline) functionality and that cannot be implemented via Java.
  - This resolves the conflicting needs: of keeping all chromium code SDK/NDK clean, yet using certain internal Android platform facilities to implement a backwards compatible and high-performance framework custom View class.
  - To minimize ABI interdependencies (e.g. avoid STL types being passed over .so boundary, incompatible new/delete calls across boundaries, etc) the interfaces declared here use simple C-style calling conventions (POD structs and static methods, injected via tables of simple function pointer).
  - The plat_support module in frameworks/webview provides the canonical concrete realization of the abstract interfaces declared here.
- android_webview/javatests
  - Integration tests for the org.chromium.android_webview.* APIs.
    - These tests exercise the top-level APIs used by the Android ‘glue layer’ to actually implement the WebView API, so they don’t actually test the full WebView stack,
    - They do however have access to internal APIs and state that is unavailable further up the stack.
    - Integration tests extend AwTestBase.
  - Unit tests for functionality implemented entirely in Java (for example AwLayoutSizerTest.java). Unit tests extend InstrumentationTestCase.
- android_webview/unittestjava
  - Java support code for C++ unittests. This is necessary to test any code that uses JNI, for example any logic checked into the android_webview/native layer.

ANDROID TREE

- frameworks/base -- core/java/android/webkit/…
  - Defines the public API to the android.webkit package (WebView, WebSettings, etc)
  - Declares the [hidden] abstract WebViewFactoryProvider, WebViewProvider interface that webview implementations must subclass.
  - Defines various “POD” like data types that are passed between the application and the concrete WebView implementation (e.g. WebViewTransport) and ancillary utility classes (e.g. URLUtil)
  - (Historic; pre-KK) Defines the WebViewClassic & related classes that powered the legacy webview implementation
- frameworks/webview -- chromium/java
  - often referred to as the “glue layer” this bridges between the core android framework, and external/chromium_org
  - Performs dependency inversion so frameworks/base and external/chromium_org have no interdependency on internal details of each other. This is the main entry-point on the java side for the chromium WebViewFactory
  - The goal is this should only depend on android_webview public APIs (some accidental exceptions exist, e.g. ThreadUtils, LibraryLoader etc) and not contain complex logic; just method forwarding (interface adaption) across the boundary.
  - By convention, this module also provides all the mappings from embedding application targetSdkVersion to

specific set of settings for workarounds & quirks that should be runtime enabled in the underlying chromium stack

- frameworks/webview -- - chromium/plat_support
  - provides a native support library to bind a select few internal native platform APIs to android_webview/public counterparts (e.g. GL functor, gralloc, and skia bitmap access utilities).
  - Dependency injection of code in this folder avoids any build-time dependency of external/chromium_org on non-NDK symbols.
- cts -- tests/tests/webkit/src/android/webkit/cts
  - Android Compatibility Test Suite for the android.webkit.* APIs. These tests have only access to the ‘public’ WebView API and test the WebView implementation in an configuration identical to when it’s actually running in an application.
  - There should be at least one test for every publicly exposed API in the suite. This makes sure everything is ‘hooked up’ correctly.

Chrome OS design documents section.